Components for automatic horn clause verification

Kafle, Bishoksan

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

Citation for published version (APA):
Kafle, B. (2016). *Components for automatic horn clause verification*. Roskilde Universitet.

COMPONENTS FOR AUTOMATIC HORN CLAUSE VERIFICATION

BISHOKSAN KAFLE

A dissertation presented to Roskilde University
in partial fulfillment of the requirement for the PhD degree in Computer Science
Department of People and Technology
Roskilde University

SUPERVISOR:
Prof. Dr. John P. Gallagher

LOCATION:
Roskilde
TIME FRAME:
June 2016

Dedicated to my parents Shruti and Puspa.

ABSTRACT

This thesis addresses problems in the area of automated software verification. Verification increases the reliability of software systems and our confidence in them. We address the problem using constrained Horn clauses, a fragment of first order logic, as a logical representation of programs.

We present an approach based on *abstraction refinement* to verifying sets of constrained Horn clauses; these usually represent safety properties of (imperative) programs. This thesis draws together a range of techniques, extends and combines them in a novel way so that they strengthen and reinforce each other. In particular, we combine (i) program transformation (specialisation), (ii) abstract interpretation and (iii) trace abstraction refinement into a single tool for Horn clause verification.

Finally we show, using a set of Horn clause benchmarks over the theory of linear arithmetic, that our approach is practical and it can solve many challenging verification problems.

ABSTRAKT

Denne afhandling omhandler problemer inden for automatiseret verifikation af programegenskaber. Verifikation øger pålideligheden af software-systemer og vores tillid til dem. Vores tilgang til problemet er baseret på brugen af "constrained Horn clauses" (CHC), der er en delmængde af første ordens logik, og som er en logisk repræsentation af programmer.

Vi præsenterer en teknik, som hedder "abstraktions-raffinering", for verifikation af et sæt CHC, der typisk repræsenterer sikkerhedsegenskaber af et program. Denne afhandling samler en række teknikker, samt udvider og kombinerer dem på en ny måde, så de styrker og komplementerer hinanden. Især kombinerer vi (i) programtransformationer (programspecialisering), (ii) abstrakt fortolkning og (iii) raffinering af "abstract traces" til et samlet værktøj til verifikation af CHC.

Endelig viser vi, ved hjælp af en række CHC eksempler fra linear aritmetik, at vores tilgang er praktisk og kan løse mange udfordrende verifikationsproblemer.

PUBLICATIONS

The thesis is based on the following published articles.

- Kafle, B. and Gallagher, J. P. Constraint specialisation in Horn clause verification. Sci. Comput. Prog., 2016. (In press).
- Kafle, B., Gallagher J.P. and Morales, J. F. RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. In: Chaudhuri, S., Farzan, A. (Eds.), CAV 2016, Proceedings, Part I. Vol. 9779 of LNCS. Springer, pp. 261-268.
- Kafle, B., Gallagher, J. P., Ganty, P. Solving non-linear Horn clauses using a linear Horn clause solver. In: Gallagher, J. P., Rümmer, P. (Eds.), Proceedings. HCVS@ETAPS 2016. Vol. 219 of EPTCS. pp. 33-48.
- Kafle, B. and Gallagher, J. P. Interpolant tree automata and their application in Horn clause verification. In Hamilton, G. W., Lisitsa, A., and Nemytykh, A. P., editors, Proceedings. VPT@ETAPS 2016, volume 216 of EPTCS, pp. 104-117.
- Kafle, B. and Gallagher, J. P. : Horn clause verification with convex polyhedral abstraction and tree automata-based refinement, Journal of Computer Languages, Systems & Structures (COMLAN) 2015 .
- Kafle, B., Gallagher, J. P., and Ganty, P. Decomposition by tree dimension in Horn clause verification. In Lisitsa, A., Nemytykh, A. P., and Pettorossi, A., editors, VPT 2015, volume 199 of EPTCS, pp. 1-14.
- Kafle, B. and Gallagher, J. P. Constraint specialisation in Horn clause verification. In Asai, K. and Sagonas, K., editors, Proceedings. PEPM 2015, pp. 85-90.
- Kafle, B. and Gallagher, J. P. Tree automata-based refinement with application to Horn clause verification. In D'Souza, D., Lal, A., and Larsen, K. G., editors, VMCAI 2015. Proceedings, volume 8931 of LNCS, pp. 209-226.
- Kafle, B. and Gallagher, J. P. Convex polyhedral abstractions, specialisation and property-based predicate splitting in horn clause verification. BjÃžrner, N., Fioravanti, F., Rybalchenko, A., and Senni, V., editors. Proceedings. HCVS 2014, volume 169 of EPTCS, 2014, pp. 53-67.
- Gallagher, J. P. and Kafle, B. Analysis and transformation tools for constrained Horn clause verification. TPLP, 14(4-5 (additional materials in online edition)): pp. 90-101.

How can one check a routine in the sense of making sure that it is right?
In order that the man who checks may not have too difficult a task the programmer
should make a number of definite assertions which can be checked individually,
and from which the correctness of the whole programme easily follows.

Alan M. Turing, "Checking a large routine", 1949.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my supervisor Prof. John Gallagher for his kind supervision of this thesis. Without his scientific virtue, immense knowledge, support and advices this thesis would have never been possible. I am truly appreciative for his great patience, generosity and understanding as well as the great friendship which has developed between us. He has proven to be a wonderful person, not just an outstanding supervisor but far beyond that and there is no doubt that the achievement of this thesis is as much his as mine.

My sincerest gratitude goes out to Jorge Navas at NASA Ames Research Center in California for hosting me at his centre during my PhD and introducing me to the practical side of computer science through the space mission software.

I would like to thank Roskilde University, the ENTRA project and the ICT-Energy project for having provided me with this generous scholarship throughout my PhD. My thanks goes out also to all the members of the ENTRA project including members from the project partners for their support and giving me the opportunity to broaden and refine my knowledge. Special thanks goes to the ENTRA Roskilde team Maja, Nina, Morten, Mads, Li and John for many useful discussions and traveling. To my good friend Maja who has been not only a great officemate but also one of the best lunch mates I have ever had! To the ever so kind and gentle Nina for having made my time spent at the office a more pleasant one. I am glad to have met a caring person such as her. I am grateful as well to the members of my department and the integrants of the research group who welcomed me and made me feel at home within the group, providing me with the required platform to work, develop and polish my ideas. I am also thankful to Kija whose friendship has helped me through some very hard times. To Irina, Moten B., Benedicte, Maria, Junia and other PhD students from the department for their support and companionship provided throughout the course. Special thanks to my very good friend and colleague Mostafa for a wonderful time. I thank him for being a true friend in all level. It is a real pleasure to have crossed paths with someone like him whose help during my hard time is unforgettable.

I could never forget to also thank my longtime friends Aroj, Subash and Luc who have supported me in their own special ways and accompanied me on this voyage despite being miles away. Many thanks for everything they have done and continue to do in the name of brotherhood and friendship. Thanks to all my friends from Danmarks Internationale Kollegium, the place I called home during my PhD. They are not just good friends but brothers and sisters from another mother.

To all those I have not mentioned who have helped me in one way or another during this important moment in my life. Thank you from the bottom of my heart.

To my co-authors Pierre Ganty and Josè F. Morales for their invaluable inputs and indispensable contributions made while working together with me over the course of this PhD. Their intellectual wealth so generously shared has helped carved my own modus operandi and I hope that we will continue to collaborate at even greater depths in the future. To Patrick Blackburn for his time spent proof reading my thesis and ensuring that I remained focused and motivated when the going got tough. His encouragement and interesting conversations kept me sane and simply saying thanks will never suffice to replay the huge debt I have with him.

I would like to thank the assessment committee for taking the time to read and comment on this dissertation. I am grateful for their time and their valuable feedback.

Finally, I would like to extend my heartfelt gratitude to my beloved parents, sisters and family to whom I am so immensely honoured to dedicate this thesis. They have been one true consistent force of support and trust throughout this journey and their unwavering faith in my ability to succeed provided me with the strength I so needed to persevere. In God I remain.

Roskilde, June 2016

Contents

List of Figures

List of Tables

ACRONYMS

CHC Constrained Horn Clause

CLP Constraint Logic Programming

FOL First Order Logic

CPA Convex Polyhedral Analysis

FTA Finite Tree Automata

SMT Satisfiability Modulo Theories

PE Partial Evaluation

CEGAR CounterExample Guided Abstraction Refinement

1 INTRODUCTION

This thesis addresses problems in the area of automated software verification. Software systems are used intensively in all sectors of our life and their correct functioning is essential in many cases. Verification increases the reliability of software systems and our confidence in them. Manual verification is an error-prone and tedious task, especially due to the increase in size and complexity of software systems, and thus its automation is highly desirable.

An automated software verifier is an algorithm that checks whether a given program satisfies a given property. This general problem is undecidable; that is, it is not possible to devise an algorithm that terminates and correctly decides whether a program satisfies a (non-trivial) semantic property or not. Even for decidable cases of the general problem such as the verification of programs having finite state spaces, the problem can be computationally intractable [86]. A great deal of effort has therefore been spent in finding algorithms that verify as many program properties as possible, even though there is no hope of solving the general problem. The work described in this thesis is a contribution to this effort.

Software verification ensures whether a software has a given desirable property or not. Some properties are related to avoiding runtime errors or crashes, for example, the absence of division by zero, overflow, and array out of bounds access. Other properties relate to the correctness of the program with respect to a specification. Broadly speaking, properties are classified as safety properties ("bad things" never happen) and liveness properties ("good things" eventually happen) [86]. Our focus in this thesis is on verifying safety properties of programs.

1.1 VERIFICATION USING CONSTRAINED HORN CLAUSES

In this thesis, we address the problem of automated software verification using constrained Horn clauses (CHCs), a fragment of first order logic (FOL), as a logical representation of programs. FOL is often used to formalise the semantics of programs. The meaning of a program P is represented by a formula in FOL and a model of the formula represents the runtime state space of P. A program property to be verified is also represented in FOL. In this way the verification problem can be viewed as checking satisfiability of FOL formulas. CHCs, although forming a fragment of FOL, have been shown to be a powerful and flexible representation for specifying the syntax and the semantics (small- and big-step operational semantics, denotational semantics etc.) of a variety of programming languages (imperative, functional, concurrent, etc.).

1.2 VERIFICATION TECHNIQUES

Program verification usually refers to verification of source programs, rather than some intermediate semantic form. Recently, constrained Horn clauses have been advocated as a "lingua franca" for program verification [38, 59, 65, 70, 17, 126] due to their expressiveness and well understood logical and computational properties. Instead of devising verification procedures for each source language (which is a difficult process) the aim is to devise a verification procedure for CHCs and then translate source languages into it, saving time and effort. This approach offers a clean separation of concerns: the translators deal with the programming language syntax and semantics whereas the verifiers deal with the verification procedures based on pure logic formulas.

Modern software verification techniques in the literature exploit several complementary techniques that are relevant for automatic software verification (sometimes called software model checking) such as abstract interpretation, program transformation, counterexample guided abstraction refinement, satisfiability checking modulo theories and automata-theoretic techniques (for example trace abstraction refinement).

Given the undecidability of the general verification problem, abstraction is a key technique, allowing "one-sided" program analysis. An abstraction provides sufficient conditions for deciding that a program has a certain property. Thus (i) if the sufficient condition can be proved then the program definitely has the property; however (ii) if the sufficient condition cannot be proven, then the answer "don't know" is returned even though the property may hold.

To deal with the "don't know" problem, techniques have been proposed in the literature, which refine abstractions. One such technique is counterexample guided abstraction refinement [23], which has been successfully pursued in the software verification community. The abstraction refinement process can be repeated until the required properties are verified, a genuine counterexample to the property is found or resources are exhausted. The study of abstract interpretation [32] focusses more on the design of suitable abstractions, tuning precision and scalability of analysis, but the "don't know" answer is unavoidable.

This thesis combines the practices in software verification and constraint logic programming [81, 82] (logic programming [119] extended to handle constraints over some domain) communities in new ways, taking advantage of the chosen logical form for program verification – combining strengths of powerful techniques such as program specialisation, abstract interpretation and trace abstraction refinement.

1.3 PROBLEM STATEMENT: RESEARCH GOAL

An approach for proving properties (termination, correctness, equivalence, *etc.*) of a computer program is to transform these problems into equivalent problems in FOL to be able to reason about them in a formal and rigorous manner. We have chosen CHC, a fragment of FOL for the purpose of verifying correctness of a program. The problem is to check whether, starting from some *initial configuration*, the execution of the program *Prog* leads to some *error configuration* or not. Let $\phi_{\mathtt{init}}$ and $\phi_{\mathtt{error}}$ be the formulas encoding the *initial configuration* and the *error configuration* respectively. Then the verification problem corresponds to the Hoare notion

of partial correctness specified by the Hoare triple $\{\phi_{init}\}$ *Prog* $\{\neg\phi_{error}\}$. The correctness triple can be translated into CHCs [39, 126] and the resulting set of CHCs is also called the *verification conditions*. Manna and Pnueli [122, Theorem 1] have shown that proving partial correctness of *Prog* with respect to ϕ_{init} and ϕ_{error} corresponds to checking the satisfiability of the *verification conditions* generated.

Therefore, given a set of CHCs P which are verification conditions for some safety properties, the CHC verification problem is to check whether there exists a model of P. This amount to checking the satisfiability of P. In terms of safety, P is considered *safe* (satisfiable) or *unsafe* (unsatisfiable) respectively if P has or does not have a model. In this thesis, we use constraint logic programming (CLP) program and CHCs interchangeably and a set of CHCs is also called a program. The literature in software verification, as well as in CLP, contains a wide range of techniques for checking the satisfiability of Horn clauses. These include:

PROGRAM TRANSFORMATION (PROGRAM SPECIALISATION): Program transformation is a method of manipulating a program's text by applying semantics preserving rules. Program specialisation (with respect to a goal), a source-to-source program transformation, is a semantics preserving transformation of Horn clauses whose goal is to remove program parts that are not relevant to the properties. It improves performance and precision of program analysis and can also be used as a proof technique in itself. Often it can cause a blow-up of the size of the specialised program, relative to the original.

ABSTRACT INTERPRETATION: This is a scalable program analysis technique which computes invariants. This allows many safe programs to be verified, but suffers from false alarms; unsafe programs and safe but not provably safe programs may be indistinguishable. Invariants are properties that hold at some program points. For example, an invariant could be that the value of some arithmetic expression over program variables is always positive at a given point. The balance between precision and scalability in abstract interpretation can be obtained by a suitable selection of an abstract domain.

TRACE ABSTRACTION REFINEMENT: This is an automata theoretic program analysis technique capable of dealing with program traces. A set of program traces (derivations) is abstracted and represented by an automaton which accepts a superset of the set of all error traces of the program. This is called trace abstraction. An accepting trace is picked from the automaton. If this trace is infeasible (unsatisfiable) with respect to the semantics of the programming language, it is eliminated from the program automaton using automata theoretic operations. This is called a refinement of trace abstraction. Eliminating a single trace in each abstraction refinement step may not be a good strategy if this technique is used alone as a proof technique for verification, since there can be infinitely many traces. So we can get more out of it, if we use this technique in combination with others or discover some heuristics for trace generalisation.

GOAL OF THE THESIS: We aim to bring together a range of techniques, extend and combine them in such a way that they strengthen and reinforce each other – leading to a scalable and

modular tool. In particular, this thesis extends and combines techniques such as (i) program transformation, (ii) abstract interpretation and (iii) trace abstraction refinement into a single tool for Horn clause verification and applies it to software verification problems.

1.4 CONTRIBUTIONS OF THE THESIS

This thesis advances the state of the art by drawing together a range of techniques, extending and combining them in a novel way so that they strengthen and reinforce each other. In particular, we combine (i) program transformation, (ii) abstract interpretation and (iii) trace abstraction refinement into a single tool for Horn clause verification. The combination brings out the synergy between the components. More specifically:

- Program transformation helps program analyses such as abstract interpretation in several ways: (i) by propagating constraints throughout the program; (ii) by removing some program parts irrelevant to the properties in question; and (iii) by splitting program predicates which allows simulating disjunctions using a solver handling only conjunctions of atoms. This can also be used either as a proof technique in itself or as a preprocessor for Horn clauses.
- Abstract interpretation infers program invariants, which can be used (i) to specialise clauses during program transformation; (ii) to prove program properties; (iii) to construct a trace abstraction of Horn clauses derivations.
- Trace abstraction refinement is used to refine abstract interpretation, more precisely, the traces used during abstract interpretation, by eliminating spurious traces (traces which are infeasible with respect to the semantics of CHCs) through automata theoretic operations. Such refinement of traces induces refinement of CHCs on which abstract interpretation is applied. We move from automata representation to Horn clauses to take advantage of the latter representation for specialisation and abstract interpretation.

This leads to an *abstraction refinement* framework for Horn clause verification. A schematic overview of this framework and the architecture of the RAHFT tool based on this idea is shown in Figure 1.1. It consists of two modules, namely, *Abstraction* (green box) and *Refinement* (red box). The input is a set of CHCs $\mathtt{P}$ written in CLP form and the output is *safe* (*unsafe*) if $\mathtt{P}$ is safe (unsafe). It consists of the following components.

- *Pre-processor*. A given set of CHCs $\mathtt{P}$ is specialised, first producing another set of CHCs $\mathtt{P}'$ which is equisatisfiable to $\mathtt{P}$. It is useful in propagating constraints in the program or removing parts of the program not relevant to the property in question.
- *Abstract interpreter*. $\mathtt{P}'$ is analysed using the technique of *abstract interpretation*, producing an over-approximation of the set of derivations (represented by an automaton) or the minimal model (the set of ground facts derivable from $\mathtt{P}'$) of $\mathtt{P}'$. These derivations are called a trace abstraction and the over-approximation is called a state abstraction.

- *Verifier*. These derivations or over-approximation of the minimal model are analysed using a light weight verifier (using the formulations described in Chapter 2) which checks whether the derivations include an error derivation or the model includes *false*. If the inclusion in either case is negative, then P is proven *safe*. But if there is an error derivation and if it is feasible in P, then P is *unsafe*.

- *Finite tree automata manipulator*. If the error derivation is infeasible, we eliminate this trace from the set of traces using automata theoretic operations – obtaining a refined set of traces. This is called trace refinement.

- *Clause generator*. From these refined traces and P′ we generate a new program using the connection between Horn clause derivations and finite tree automata.

This new program is again fed to the pre-processor. This process is repeated until a set of clauses is proven *safe*, *unsafe* or resources are exhausted.

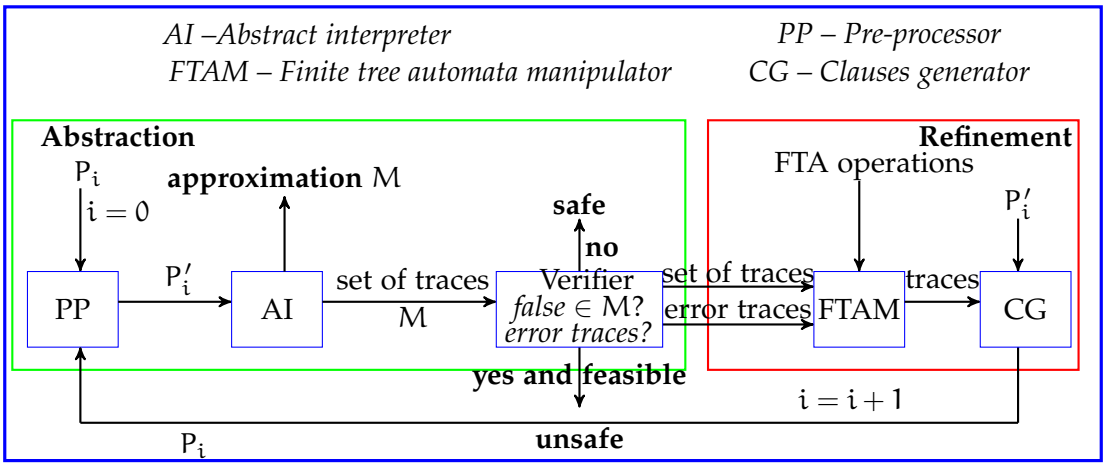

Figure 1.1: *The overview of our verification framework and the architecture of* RAHFT.

The main contributions of the thesis are the following.

- We present a satisfiability preserving transformation (specialisation) of Horn clauses. It specialises the constraints in Horn clauses with respect to a goal. The constraints in the clauses are strengthened using the invariants computed by abstract interpretation for their query-answer transformation (a program transformation which allows the query-dependence to be "compiled" to the program clauses). The effect is to propagate the constraints from the goal and from the constrained facts (clauses whose body only contain constraints). The approach is independent of the abstract domain and the constraint theory underlying the clauses and does not unfold the clauses at all; this provides benefits over other transformational approaches based on partial evaluation and fold-unfold transformation.

- We present tree-automata techniques for the refinement of abstract interpretation in Horn clause verification. We extend previous work on refining trace abstractions in several directions. Firstly, the use of tree automata allows us to capture traces in any Horn clause derivation not only in transition systems. Secondly, we show how abstract interpretation can be used to abstract Horn clause derivations (traces) in the form of finite tree automata. Thirdly, we exploit the connection between Horn clauses and finite tree automata for Horn clause transformation. Fourthly, we integrate interpolant tree automata to generalise error traces during refinement of trace abstraction. An interpolant between two unsatisfiable formulas A and B is a formula over the common variables of A and B which concisely explains why they are unsatisfiable.

- We propose a proof decomposition technique for Horn clauses, based on a characterisation of its derivation trees (the concept of *tree dimension* which is a measure of its non-linearity). A proof of a set of CHCs can be decomposed into several proofs for different values of *tree dimension*, which can be computed in parallel. In this way, the proof for the original set of CHCs can be composed from the proofs of its constituents.

- We present an *abstraction-refinement* approach for solving a set of non-linear Horn clauses (clauses which contain more than one non-constraint atom in their bodies) using an off-the-shelf linear Horn clause solver. The approach is based on a linearisation of a dimension-bounded set of Horn clauses (Horn clauses whose derivation trees have bounded *tree dimension*) using partial evaluation and the use of a linear Horn clause solver. In this, we explore the relation between the solvability of a problem and its dimension.

- The ideas presented in the thesis are implemented in RAHFT, an *abstraction refinement* tool for verifying safety properties of programs expressed as Horn clauses. This tool combines three powerful techniques for program verification in the same framework: (i) program specialisation, (ii) abstract interpretation, and (iii) trace abstraction refinement. RAHFT compares favourably with the other state of the art Horn clause verification tools in the literature. The tool has been evaluated by the artifact evaluation committee (AEC) of the CAV'16 conference and has received the seal of AEC-CAV'16.

1.5 OVERVIEW OF THE THESIS

The thesis is organised as follows: Chapter 2 provides some background on constrained Horn clauses and proof techniques for Horn clauses. Chapters 3 to 9 contain published research papers as distinct chapters. For simplicity, the title of the chapter is chosen to be the same as the title of the paper. Finally Chapter 10 concludes the thesis. A schematic overview of the thesis is shown in Figure 1.2. It summarises how the chapters fit together in the thesis and contribute to providing solutions to the problem statement (research goal). In more detail:

CHAPTER 3 Since Horn clause verification is interesting to both the logic programming and the verification communities [16], several techniques and tools have been developed for

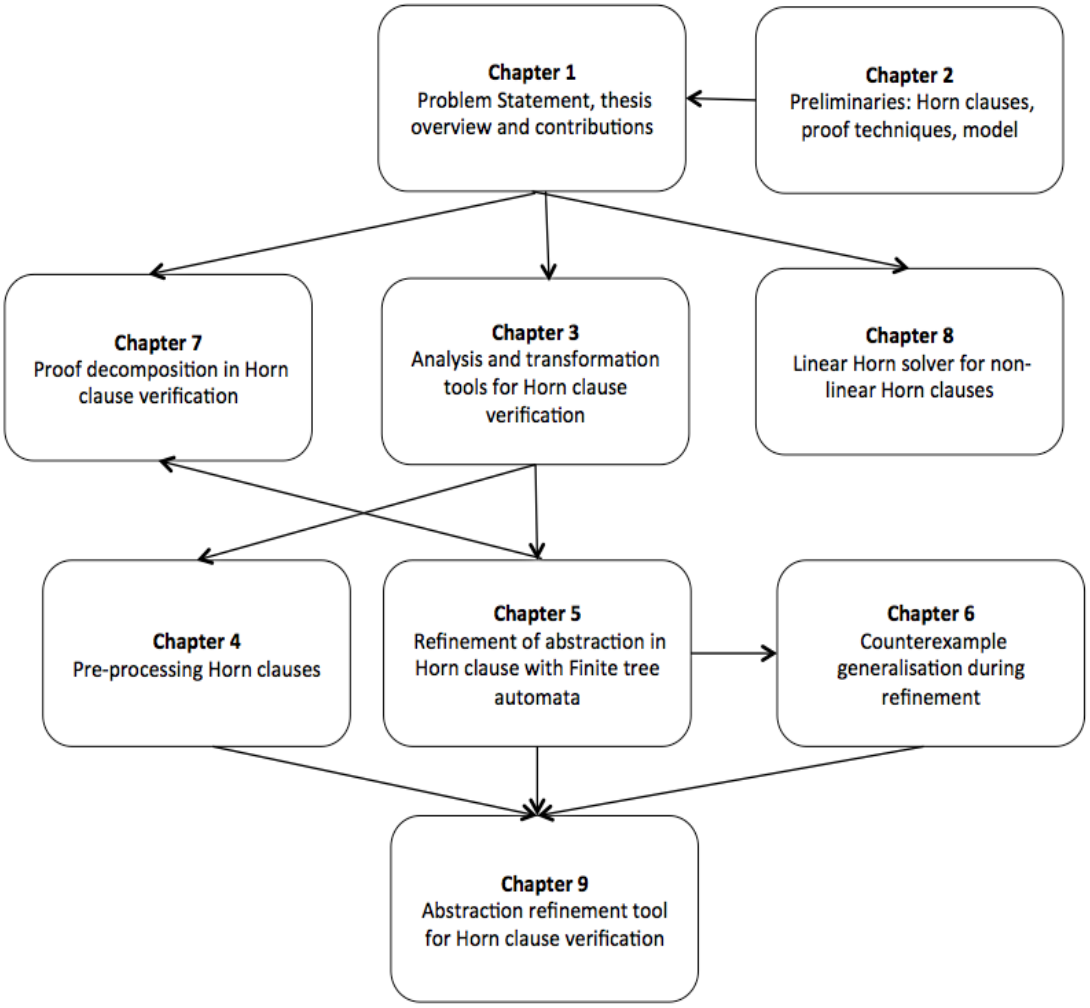

Figure 1.2: Graphical representation of the thesis structure. The arrows represent dependencies between the chapters.

it. In Chapter 3, we study the role of off-the-shelf techniques from the literature in analysis and transformation of CLPs and their combination to solve CHC verification problems. Our findings (i) show that a suitable combination of tools based on well-known techniques such as abstract interpretation, program specialisation and query-answer transformations can often solve many problems; and (ii) provide insights into the design of automatic CHC verification tools based on a library of components. This chapter provides a basis for the verification framework presented in Section 1.4, which has been improved and strengthened throughout the thesis. The chapters that follow contribute to each component of the framework. This chapter is identical to the paper [59].

CHAPTER 4 Our study in Chapter 3 suggested a prominent role for program transformation in Horn clause verification, and its role as a pre-processing component in our verification framework could be an advantage. This is also motivated by the recent advances in Satisfiability Modulo Theories (SMT) solving and transformational approaches to Horn clause verification. Therefore, we developed a method for program transformation which specialises the constraints in Horn clauses with respect to a goal. The method uses abstract interpretation and query-answer transformation to program specialisation. The method is generic in the sense that it is independent of the abstract domain and the constraints theory underlying the clauses. This chapter is essentially the same as the paper [98] which in turn was based on the papers [94, 93].

CHAPTER 5 Abstract interpretation as a scalable program analysis technique allows many true properties to be proven and can be used as an analysis component in our framework as indicated in Chapter 3. However it suffers from *false alarms*: that is, false properties and true but not provable properties are indistinguishable. To mitigate this problem we resorted to a refinement technique using finite tree automata (FTAs), described in Chapter 5. The motivations behind this are: (i) FTAs provide a clear method for manipulating Horn clause derivations which allows us to refine a set of traces rather than a set of states; (ii) these trace refinements induce refinement in the original program; (iii) the refinements are property directed; and (iv) the refinement is independent of the abstract domain and constraint theory underlying the Horn clauses. This chapter is essentially the same as the paper [96] which in turn was based on the paper [95].

CHAPTER 6 The refinement using FTAs presented in Chapter 5 eliminates one spurious trace in each iteration of the abstraction refinement loop in our framework, though the FTA operations offer removal of a set of traces without any additional implementation cost. Therefore we use the concept of *interpolant tree automata* to discover more infeasible traces by trace generalisation and remove them in a single abstraction refinement iteration. This was motivated by the concept of *interpolant automata* [75, 147] and the success of interpolation based techniques in Horn clause verification [126, 65]. This chapter is essentially the same as [97].

CHAPTER 7 In the previous chapters, we looked for a single proof for a set of Horn clauses. In this chapter, we propose a proof decomposition technique for a set of Horn clauses, based

on a characterisation of its derivation trees (the concept of *tree dimension* which is a measure of its non-linearity). A proof of a set of CHCs can be decomposed into the separate proofs of a set of sets of CHCs using *tree dimension*. Then the proof for the original set of CHCs can be obtained from the proof of its constituents, which could be done in parallel. This chapter is essentially the same as the paper [99].

CHAPTER 8 In the literature, there are Horn clause solvers whose underlying engine can handle only linear clauses; in principle this restricts their applicability. Therefore the goal of this chapter is to provide a method which allows a linear solver to solve non-linear problems. The results suggest that it is feasible. As a consequence, more dedicated linear solvers can be developed which can be more efficient and simpler than the non-linear ones. This chapter is essentially the same as the paper [101].

CHAPTER 9 Here we present RAHFT (Refinement of Abstraction in Horn clauses using Finite Tree automata), an *abstraction refinement* tool for verifying safety properties of programs expressed as Horn clauses. This tool encapsulates the ideas presented in previous chapters, and combines three powerful techniques for program verification in the same framework: (i) program specialisation, (ii) abstract interpretation, and (iii) trace abstraction refinement. Its modular design and customizable components allows for experimentation with new verification techniques and tools developed for Horn clauses. This chapter is essentially the same as the paper [103].

CHAPTER 10 Concludes the thesis by highlighting the contributions and indicating some future directions of work.

2

PRELIMINARIES

This chapter aims to introduce readers to Horn clauses and different proof techniques for checking satisfiability of a set of Horn clauses. Since the thesis draws on several techniques in the literature and is based on a set of self-contained papers, we omit detailed background materials and refer to standard textbooks and articles. The thesis does presuppose background knowledge in several areas such as first order logic, logic programming, abstract interpretation and so on. So, for interested readers who would like to know more, we suggest different levels of reading materials: introductory material, advanced material and key research papers in Appendix A.

2.1 SYNTAX AND SEMANTICS OF CONSTRAINT LOGIC PROGRAMS

We start by recalling some basic notions and terminology of CLP. We consider CLP programs parametrized by a constraint theory D and represent it by CLP(D) [81]. A signature Σ of (first order) predicate logic consists of a finite or countably infinite set $\mathcal{R}$ of predicate (relation) symbols with arity $n \geqslant 0$, a finite or countably infinite set $\mathcal{F}$ of function symbols with arity $n \geqslant 0$, a countably infinite set $\mathcal{V}$ of variables, the set $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$ of connectives, and the set $\{\forall, \exists\}$ of quantifiers. We denote by $\mathcal{L}(\mathcal{R}, \mathcal{F}, \mathcal{V})$ (or simply $\mathcal{L}$), the language generated by the signature. We represent the set of function and predicate symbols defined in the constraint domain by $\mathcal{F}_c$ and $\mathcal{R}_c$ respectively and the set of function and predicate symbols defined by the user by $\mathcal{F}_u$ and $\mathcal{R}_u$ respectively.

A *term* is either a variable in $\mathcal{V}$ or an expression of the form $f(t_1, \ldots, t_n)$ where $f/n \in \mathcal{F}$ and $t_1, \ldots, t_n$ are terms. A term is *closed* or *ground* (*instantiated*), if it does not contain any variable.

An *atomic formula* (*atom*) of the language $\mathcal{L}(\mathcal{R}, \mathcal{F}, \mathcal{V})$ is an expression of the form $p(t_1, \ldots, t_n)$, where $p/n \in \mathcal{R}_u$ and $t_1, \ldots, t_n$ are terms.

A predicate logic formula of the language $\mathcal{L}(\mathcal{R}, \mathcal{F}, \mathcal{V})$ is either an atomic formula or a formula constructed from already constructed formulas using the connectives and quantifiers [49]. A formula is closed if all variable occurrences in the formula are within the scope of a quantifier over the variable.

An *atomic constraint* is an atomic formula $p(t_1, \ldots, t_n)$, where $p/n \in \mathcal{R}_c$ and $t_1, \ldots, t_n$ are terms. A constraint is either *false*, or *true* or a first order formula built from atomic constraints.

A CLP program P is a finite set of clauses of the form $H \leftarrow \phi, B_1, \ldots, B_k$ where H is an atom and B_i's are atoms or constraints. We implicitly assume that all the variables in the clauses are universally quantified. Sometimes we represent the clause by $H \leftarrow \phi, B$, where ϕ is a conjunction of constraints and B is a (possibly empty) conjunction of atoms. H is called the head and ϕ, B is called the body of the clause. A *constrained fact* is a clause of the form $H \leftarrow \phi$. If ϕ is *true* then the constrained fact is called a *fact* and is usually written as H.

The semantics of CLP is parametrized by the interpretation of constraints. A constraint interpretation consists of a non-empty set $\mathcal{D}$ (domain of interpretation) and a mapping $\_^I$ (indicated by a super-script) which satisfies the following conditions.

- Every n-ary function symbol $f/n \in \mathcal{F}_c$ is mapped to an n-ary function $f^I : \mathcal{D}^n \to \mathcal{D}$.
- Every n-ary relation symbol $p/n \in \mathcal{R}_c$ is mapped to an n-ary relation $p^I \subseteq \mathcal{D}^n$.

Let $\mathcal{H}$ be the set of all possible ground terms built from the elements of $\mathcal{D}$ and $f \in \mathcal{F}_u$ in the language of a program P. Given a constraint interpretation D, an interpretation of the predicate symbols in $\mathcal{R}_u$ is called a D-*interpretation* and is defined as follows. A D-*interpretation* is an interpretation with universe $\mathcal{H}$ such that: (i) it assigns to elements of $\mathcal{F}_c$ and $\mathcal{R}_c$ the meanings given above, and (ii) it is the *Herbrand interpretation* [119] for function and predicate symbols in $\mathcal{F}_u$ and $\mathcal{R}_u$. A D-*interpretation* which makes true (satisfies) every clause (formula) in P is called a D-*model*. An interpretation can be seen as a set of ground facts of the form $p(d_1, \ldots, d_k)$ where $d_i \in \mathcal{H}$.

The semantics of a CLP program P is defined to be the least D-*model* of P, that is, the least D-*interpretation* that satisfies each clause of P. This is also called the least *model*, denoted $M[\![P]\!]$ [81]. From now on we omit D from the interpretation and model and assume it implicitly.

We refer to [81] for the interpretation of CHCs as CLP programs. A set of CHCs can be regarded as a CLP program. CHCs and CLP serve as specifications or semantic representations of programs or design models. In contrast to CHCs, a CLP program is an executable program. However the semantic equivalence of CHC and CLP allows techniques developed in one framework to be applied to the other.

2.2 CONSTRAINED HORN CLAUSES: INTERPRETATIONS AND MODELS

A (constrained) Horn clause is a CLP clause. Using FOL notation, we write it as $\forall(\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k) \rightarrow p(X))$ $(k \geqslant 0)$, where ϕ is a constraint in some constraint theory, $X_1, \ldots, X_k, X$ are (possibly empty) vectors of terms, $p_1, \ldots, p_k, p$ are predicate symbols. In this thesis, we assume that the arguments of a predicate are always regarded as a tuple of distinct variables unless otherwise stated; when we write $p(X)$ or $p(a)$, then X and a stand for (possibly empty) tuples of variables and ground terms respectively. Following the syntactic conventions of CLP [81] we write a Horn clause as $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$. We consider that ϕ is a conjunction of atomic constraints without loss of generality. This is because a constraint can be written using only the connectives $\{\wedge, \vee, \neg\}$; the constraint theory we consider is the theory of linear arithmetic which is closed under negation; and any clause of the form $H \leftarrow (\phi_1 \vee \phi_2) \wedge B$ can be replaced by the clauses $H \leftarrow \phi_1 \wedge B$ and $H \leftarrow \phi_2 \wedge B$. An *integrity constraint* is a special kind of CHC whose head is *false* where the predicate *false* is always interpreted as the empty relation FALSE. From now on we assume that $\mathcal{F}_u = \emptyset$.

An interpretation of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \phi$ where A is an atomic formula $p(Z)$ and Z is a tuple of distinct variables and ϕ is a constraint over Z (quantifier free linear arithmetic formula). The constrained fact $A \leftarrow \phi$

stands for the set of ground facts $A\theta$ (where θ is a grounding substitution) such that $\phi\theta$ holds in the constraint theory. The interpretations of a program are related by a subset relation. We write $M_1 \subseteq M_2$ if the set of denoted ground facts of M_1 is contained in the set of denoted ground facts of M_2. An interpretation M satisfies a CHC $p_0(X_0) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$, if M contains constrained facts $\{p_i(X_i) \leftarrow \phi_i \mid 0 \leqslant i \leqslant n\}$, and $\forall(\phi_0 \leftarrow (\phi \wedge \bigwedge_{i=1}^{n} \phi_i))$ is true. Note that a set of clauses without integrity constraints is satisfiable.

A set of Horn clauses P enjoys an interesting property, called the *model intersection property*. This states that given a set $\{M_j \mid j \in J\}$ of *models* of P (expressed as a set of ground facts) their intersection $\bigcap_{j\in J} M_j$ is also a model of P [81].

MINIMAL MODELS. There exists a minimal model (the intersection of all models) with respect to the subset ordering, denoted $M[\![P]\!]$ where P is the set of CHCs. The fixed point semantics (model semantics) of P are based on the immediate consequence operator (function) represented as S_P^D, which is an extension of the standard T_P operator from logic programming [119], extended to handle the constraint domain D [81, Section 4]. It is defined on sets of facts, which forms a complete lattice under the subset ordering. By the Knaster-Tarski theorem, there exists a least and a greatest fixed points of S_P^D since S_P^D is a monotone function on a complete lattice. $M[\![P]\!]$ can be computed as the least fixed point (lfp) of S_P^D. Furthermore $\mathsf{lfp}(S_P^D)$ can be computed as the limit of the ascending sequence of interpretations $\emptyset, S_P^D(\emptyset), S_P^D(S_P^D(\emptyset)), \ldots$. We refer to [81] for further details.

2.3 PROOF TECHNIQUES FOR CONSTRAINED HORN CLAUSES

The CHC verification problem is to check whether there exists a model of P. If so we say that P is safe. We write $P \models F$ to mean that F is a logical consequence of P, that is, that every interpretation satisfying P also satisfies F. Similarly $P \vdash F$ means that F is derivable from P using some proof procedure. For a theory with a sound and complete proof procedure, for example FOL, we can replace $\models$ by $\vdash$ and vice-versa. We define several *proof techniques* for Horn clauses verification; depending on the choice made, we exploit different formulations for Horn clause verification. These are based on two equivalent formulations [81] for CHC verification, namely, Formulations 2.1 and 2.2.

Formulation 2.1 (Model based)**.** *P has a model if and only if $P \not\models$ false*

Formulation 2.2 (Deductive or proof based)**.** *P has a model if and only if $P \not\vdash$ false (false is not provable from P).*

PROOF BY SPECIALISATION. A specialisation of a set of CHCs P with respect to an atom A is the transformation of P to another set of CHCs P' such that $P \models A$ if and only if $P' \models A$. Specialisation is usually viewed as a program optimisation method possibly removing redundancy and pre-computing statically determined computations of a general purpose program. Formulations 2.1 or 2.2 are equally relevant for this techniques. In particular, we have the following formulation.

Formulation 2.3 (Specialisation based)**.** *Let* P *be a set of CHCs, and* P_s *be a specialised set of CHCs with respect to the atom false.* P *has a model if* P_s *contains no clause with false head.* P *has no model if* P_s *contains a clause of the form false* $\leftarrow$ *true.*

We exploit Formulation 2.3 in Chapter 4.

PROOF BY OVER-APPROXIMATION OF THE MINIMAL MODEL. Given a set of CHCs, its minimal model $M[\![P]\!]$ is equivalent to the set of atomic consequences of P [119]. That is, $P \models p(a)$ if and only if $p(a) \in M[\![P]\!]$. It is sufficient to find a set of constrained facts M' such that $M[\![P]\!] \subseteq M'$, where *false* $\notin M'$. We have:

Formulation 2.4 (Over-approximation of the minimal model)**.** *Given* P *and* M' *such that* $M[\![P]\!] \subseteq M'$. P *has a model if false* $\notin M'$.

Formulation 2.4 forms the basis of tools based on *abstract interpretation* or *predicate abstraction* [65, 70, 96]. We exploit Formulation 2.4 in Chapter 5. But nothing can be said if *false* $\in M'$. This could mean either P has no model, or M' was too large.

PROOF BY OVER-APPROXIMATION OF THE SUCCESSFUL DERIVATIONS OF P. The CHC verification problem for P is equivalent to checking that there are no successful derivations of *false* using P. If we can find a superset of successful derivations of P which does not contain any derivation of *false*, then we have shown that P has a model.

Formulation 2.5 (Over-approximation of successful derivations of P (trace based))**.** *Let* P *be a set of CHCs,* $\mathfrak{I}$ *a set of successful derivations of* P, $\mathfrak{I}'$ *a set of derivations such that* $\mathfrak{I} \subseteq \mathfrak{I}'$. P *has a model if* $\forall t.(t \text{ rooted at } \mathit{false} \wedge \text{ successful } t \rightarrow t \notin \mathfrak{I}')$.

Formulation 2.5 forms the basis of the tools described in [74, 147], which we exploit in Chapter 5. $\mathfrak{I}'$ is also called a *trace-abstraction* for P. If $t \in \mathfrak{I}'$ is a trace rooted at *false* and is successful with respect to P, then we say that P has no model and the derivation t is a counterexample which justifies why P has no model.

Given a Horn clause derivation (a labelled tree), its *tree-dimension* is a measure of its non-linearity. For example a linear derivation has dimension 0 while a perfect binary tree has dimension equal to its height. Based on this notion, we have the following formulation for Horn clause verification.

Formulation 2.6 (Deductive–dimension based)**.** P *has a model if and only if there is no successful derivation of false – of any dimension.*

We exploit this formulation in Chapter 7 to decompose the verification problem.

3

ANALYSIS AND TRANSFORMATION TOOLS FOR CONSTRAINED HORN CLAUSE VERIFICATION

With John P. Gallagher

Abstract
Several techniques and tools have been developed for verification of properties expressed as Horn clauses with constraints over a background theory (CHC). Current CHC verification tools implement intricate algorithms and are often limited to certain subclasses of CHC problems. Our aim in this work is to investigate the use of a combination of off-the-shelf techniques from the literature in analysis and transformation of Constraint Logic Programs (CLPs) to solve challenging CHC verification problems. We find that many problems can be solved using a combination of tools based on well-known techniques from abstract interpretation, semantics-preserving transformations, program specialisation and query-answer transformations. This gives insights into the design of automatic, more general CHC verification tools based on a library of components.

Keywords: Constraint Logic Program, Constrained Horn Clause, Abstract Interpretation, Software Verification.

3.1 INTRODUCTION

CHCs provide a suitable intermediate form for expressing the semantics of a variety of programming languages (imperative, functional, concurrent, *etc.*) and computational models (state machines, transition systems, big- and small-step operational semantics, Petri nets, *etc.*). As a result it has been used as a target language for software verification. Recently there is a growing interest in CHC verification from both the logic programming and software verification communities, and several verification techniques and tools have been developed for CHC.

CLPs are syntactically and semantically the same as CHC. The main difference is that sets of constrained Horn clauses are not necessarily intended for execution, but rather as specifications. From the point of view of verification, we do not distinguish between CHC and pure CLP. Much research has been carried out on the analysis and transformation of CLP programs, typically for synthesising efficient programs or for inferring run-time properties of programs for the purpose of debugging, compile-time optimisations or program understanding. In this chapter we investigate the application of this research to the CHC verification problem.

In Section 3.2 we define the CHC verification problem. In Section 3.3 we define basic transformation and analysis components drawn from or inspired by the CLP literature. Section 3.4

discusses the role of these components in verification, illustrating them on an example problem. In Section 3.5 we construct a tool-chain out of these components and test it on a range of CHC verification benchmark problems. The results reported represent one of the main contributions of this work. In Section 3.6 we propose possible extensions of the basic tool-chain and compare them with related work on CHC verification tool architectures. Finally in Section 3.7 we summarise the conclusions from this work.

3.2 BACKGROUND: THE CHC VERIFICATION PROBLEM

A CHC is a first order predicate logic formula of the form $\forall(\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k) \rightarrow p(X))$ $(k \geqslant 0)$, where ϕ is a conjunction of constraints with respect to some background theory, X_i, X are (possibly empty) vectors of distinct variables, $p_1, \ldots, p_k, p$ are predicate symbols, $p(X)$ is the head of the clause and $\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k)$ is the body. Sometimes the clause is written $p(X) \leftarrow \phi \wedge p_1(X_1), \ldots, p_k(X_k)$ and in concrete examples it is written in the form `p(X) :- `ϕ`, `$p_1(X_1)$`,...,`$p_k(X_k)$. In examples, predicate symbols start with lowercase letters while we use uppercase letters for variables.

We assume here that the constraint theory is linear arithmetic with relation symbols $\leqslant, \geqslant, >, <$ and $=$ and that there is a distinguished predicate symbol *false* which is interpreted as FALSE. We assume that the predicate *false* only occurs in the head of clauses; we call clauses whose head is *false*, *integrity constraints*, following the terminology of deductive databases. Thus the formula $\phi_1 \leftarrow \phi_2 \wedge p_1(X_1), \ldots, p_k(X_k)$ is equivalent to the formula $\mathit{false} \leftarrow \neg\phi_1 \wedge \phi_2 \wedge p_1(X_1), \ldots, p_k(X_k)$. The latter might not be a CHC but can be converted to an equivalent set of CHCs by transforming the formula $\neg\phi_1$ and distributing any disjunctions that arise over the rest of the body. For example, the formula `X=Y :- p(X,Y)` is equivalent to the set consisting of the CHCs `false :- X>Y, p(X,Y)` and `false :- X<Y, p(X,Y)`. Integrity constraints can be viewed as safety properties. If a set of CHCs encodes the behaviour of some system, the bodies of integrity constraints represent unsafe states. Thus proving safety consists of showing that the bodies of integrity constraints are false in all models of the CHCs.

THE CHC VERIFICATION PROBLEM. To state this more formally, given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. We restate this property in terms of the derivability of the predicate *false*.

Lemma 3.1. *P has a model if and only if $P \not\models$ false.*

Proof. Let us write $I \models F$ to mean that interpretation I satisfies F (I is a model of F).

$$\begin{aligned} P \not\models \mathit{false} &\equiv \exists I.(I \models P \text{ and } I \not\models \mathit{false}) \\ &\equiv \exists I.I \models P \quad (\textit{since } I \not\models \mathit{false} \textit{ is } \textsc{True} \textit{ by defn. of false}) \\ &\equiv P \text{ has a model.} \end{aligned}$$

□

Obviously any model of P assigns FALSE to the bodies of integrity constraints.

The verification problem can be formulated deductively rather than model-theoretically. Let the relation $P \vdash A$ denote that A is derivable from P using some proof procedure. If the proof procedure is sound and complete then $P \not\models A$ if and only if $P \not\vdash A$. So the verification problem is to show (using CLP terminology) that the computation of the goal $\leftarrow$ *false* in program P does not succeed using a complete proof procedure. Although in this work we follow the model-based formulation of the problem, we exploit the equivalence with the deductive formulation, which underlies, for example, the query-answer transformation and specialisation techniques to be presented.

3.2.1 *Representation of Interpretations*

An interpretation of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \mathcal{C}$ where A is an atomic formula $p(Z_1, \ldots, Z_n)$ where $Z_1, \ldots, Z_n$ are distinct variables and $\mathcal{C}$ is a constraint over $Z_1, \ldots, Z_n$. If $\mathcal{C}$ is *true* we write $A \leftarrow$ or just A. The constrained fact $A \leftarrow \mathcal{C}$ is shorthand for the set of variable-free facts $A\theta$ such that $\mathcal{C}\theta$ holds in the constraint theory, and an interpretation M denotes the set of all facts denoted by its elements; M assigns true to exactly those facts. $M_1 \subseteq M_2$ if the set of denoted facts of M_1 is contained in the set of denoted facts of M_2.

MINIMAL MODELS. A model of a set of CHCs is an interpretation that satisfies each clause. There exists a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$ where P is the set of CHCs. $M[\![P]\!]$ can be computed as the least fixed point (lfp) of an immediate consequences operator, $T_P^{\mathcal{C}}$, which is an extension of the standard T_P operator from logic programming, extended to handle constraints [81]. Furthermore $\mathrm{lfp}(T_P^{\mathcal{C}})$ can be computed as the limit of the ascending sequence of interpretations $\emptyset, T_P^{\mathcal{C}}(\emptyset), T_P^{\mathcal{C}}(T_P^{\mathcal{C}}(\emptyset)), \ldots$. For more details, see [81]. This sequence provides a basis for abstract interpretation of CHCs.

PROOF BY OVER-APPROXIMATION OF THE MINIMAL MODEL. It is a standard theorem of CLP that the minimal model $M[\![P]\!]$ is equivalent to the set of atomic consequences of P. That is, $P \models p(a)$ if and only if $p(a) \in M[\![P]\!]$. Therefore, the CHC verification problem for P is equivalent to checking that *false* $\notin M[\![P]\!]$. It is sufficient to find a set of constrained facts M' such that $M[\![P]\!] \subseteq M'$, where *false* $\notin M'$. This technique is called proof by over-approximation of the minimal model.

3.3 RELEVANT TOOLS FOR CHC VERIFICATION

In this section, we give a brief description of some relevant tools borrowed from the literature in analysis and transformation of CLP.

UNFOLDING. Let P be a set of CHCs and $c_0 \in P$ be $H(X) \leftarrow \mathcal{B}_1, p(Y), \mathcal{B}_2$ where $\mathcal{B}_1, \mathcal{B}_2$ are possibly empty conjunctions of atomic formulas and constraints. Let $\{c_1, \ldots, c_m\}$ be the set of clauses of P that have predicate p in the head, that is, $c_i = p(Z_i) \leftarrow \mathcal{D}_i$, where the

variables of these clauses are standardised apart [119] from the variables of c_0 and from each other. Then the result of unfolding c_0 on $p(Y)$ is the set of CHCs $P' = P \setminus \{c_0\} \cup \{c'_1, \ldots, c'_m\}$ where $c'_i = H(X) \leftarrow \mathcal{B}_1, Y = Z_i, \mathcal{D}_i, \mathcal{B}_2$. The equalities $Y = Z_i$ stands for the conjunction of the equality of the respective elements of the vectors Y and Z_i. It is a standard result that unfolding a clause in P preserves P's minimal model [131]. In particular, $P \models \mathit{false}$ iff $P' \models \mathit{false}$.

SPECIALISATION. A set of CHCs P can be specialised with respect to a query. Assume A is an atomic formula; then we can derive a set P_A such that $P \models A \equiv P_A \models A$. P_A could be simpler than P, for instance, parts of P that are irrelevant to A could be omitted in P_A. In particular, the CHC verification problem for P_{false} and P are equivalent. There are many techniques in the CLP literature for deriving a specialised program P_A. Partial evaluation is a well-developed method [52, 111].

We make use a form of specialisation known as slicing (commonly used in the setting of imperative programs), more specifically redundant argument filtering [114], in which predicate arguments can be removed if they do not affect a computation. Given a set of CHCs P and a query A, denote by P^{raf}_A the program obtained by applying the RAF algorithm from [114] with respect to the goal A. We have the property that $P \models A$ iff $P^{\mathsf{raf}}_A \models A$ and in particular that $P \models \mathit{false}$ iff $P^{\mathsf{raf}}_{\mathit{false}} \models \mathit{false}$.

QUERY-ANSWER TRANSFORMATION. Given a set of CHCs P and an atomic query A, the query-answer transformation of P with respect to A is a set of CHCs which simulates the computation of the goal $\leftarrow A$ in P, using a left-to-right computation rule. Query-answer transformation is a generalisation of the magic set transformations for Datalog. For each predicate p, two new predicates p_{ans} and p_{query} are defined. For an atomic formula A, A_{ans} and A_{query} denote the replacement of A's predicate symbol p by p_{ans} and p_{query} respectively. Given a program P and query A, the idea is to derive a program P^{qa}_A with the following property $P \models A$ iff $P^{\mathsf{qa}}_A \models A_{ans}$. The A_{query} predicates represent calls in the computation tree generated during the execution of the goal. For more details see [41, 54, 24]. In particular, $P^{\mathsf{qa}}_{\mathit{false}} \models \mathit{false}_{ans}$ iff $P \models \mathit{false}$, so we can transform a CHC verification problem to an equivalent CHC verification problem on the query-answer program generated with respect to the goal $\leftarrow \mathit{false}$. Please refer to Chapter 4 for more details.

PREDICATE SPLITTING. Let P be a set of CHCs and let $\{c_1, \ldots, c_m\}$ be the set of clauses in P having some given predicate p in the head, where $c_i = p(X) \leftarrow \mathcal{D}_i$. Let $C_1, \ldots, C_k$ be some partition of $\{c_1, \ldots, c_m\}$, where $C_j = \{c_{j_1}, \ldots, c_{j_{n_j}}\}$. Define k new predicates $p_1 \ldots p_k$, where p_j is defined by the bodies of clauses in partition C_j, namely $Q^j = \{p_j(X) \leftarrow \mathcal{D}_{j_1}, \ldots, p_j(X) \leftarrow \mathcal{D}_{j_{n_j}}\}$. Finally, define k clauses $C_p = \{p(X) \leftarrow p_1(X), \ldots, p(X) \leftarrow p_k(X)\}$. Then we define a splitting transformation as follows.

1. Let $P' = (P \setminus \{c_1, \ldots, c_m\}) \cup C_p \cup Q^1 \cup \ldots \cup Q^k$.
2. Let P^{split} be the result of unfolding every clause in P' whose body contains $p(Y)$ with the clauses C_p.

In our applications, we use splitting to create separate predicates for clauses for a given predicate whose constraints are mutually exclusive. For example, given the clauses `new3(A,B) :- A=<99, new4(A,B)` and `new3(A,B) :- A>=100, new5(A,B)`, we produce two new predicates, since the constraints `A=<99` and `A>=100` are disjoint. The new predicates are defined by clauses $\texttt{new3}_1\texttt{(A,B) :- A=<99, new4(A,B)}$ and $\texttt{new3}_2\texttt{(A,B) :- A>=100, new5(A,B)}$, and all calls to `new3` throughout the program are unfolded using these new clauses. Splitting has been used in the CLP literature to improve the precision of program analyses, for example in [139]. In our case it improves the precision of the convex polyhedron analysis discussed below, since separate polyhedra will be maintained for each of the disjoint cases. The correctness of splitting can be shown using standard transformations that preserve the minimal model of the program (with respect to the predicates of the original program) [131]. Assuming that the predicate *false* is not split, we have that $P \models \mathit{false}$ iff $P^{\mathsf{split}} \models \mathit{false}$.

CONVEX POLYHEDRON APPROXIMATION. Convex polyhedron analysis [31] is a program analysis technique based on abstract interpretation [32]. When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P, where M' contains at most one constrained fact $p(X) \leftarrow \mathcal{C}$ for each predicate p. The constraint $\mathcal{C}$ is a conjunction of linear inequalities, representing a convex polyhedron. The first application of convex polyhedron analysis to CLP was by Benoy and King [10]. Since the domain of convex polyhedra contains infinite increasing chains, the use of a widening operator is needed to ensure convergence of the abstract interpretation. Furthermore much research has been done on improving the precision of widening operators. One technique is known as widening-upto, or widening with thresholds [72].

Recently, a technique for deriving more effective thresholds was developed [108], which we have adapted and found to be effective in experimental studies. The thresholds are computed by the following method. Let $T_P^{\mathcal{C}}$ be the standard immediate consequence operator for CHCs, that is, $T_P^{\mathcal{C}}(I)$ is the set of constrained facts that can be derived in one step from a set of constrained facts I. Given a constrained fact $p(Z) \leftarrow \mathcal{C}$, define $\mathsf{atomconstraints}(p(Z) \leftarrow \mathcal{C})$ to be the set of constrained facts $\{p(Z) \leftarrow C_i \mid \mathcal{C} = C_1 \wedge \ldots \wedge C_k, 1 \leqslant i \leqslant k)\}$. The function atomconstraints is extended to interpretations by

$$\mathsf{atomconstraints}(I) = \bigcup_{p(Z) \leftarrow \mathcal{C} \in I} \{\mathsf{atomconstraints}(p(Z) \leftarrow \mathcal{C})\}.$$

Let $I_\top$ be the interpretation consisting of the set of constrained facts $p(Z) \leftarrow \mathit{true}$ for each predicate p. We perform three iterations of $T_P^{\mathcal{C}}$ starting with $I_\top$ (the first three elements of a "top-down" Kleene sequence) and then extract the atomic constraints. That is, thresholds is defined as follows.

$$\mathsf{thresholds}(P) = \mathsf{atomconstraints}(T_P^{\mathcal{C}(3)}(I_\top))$$

A difference from the method in [108] is that we use the concrete semantic function $T_P^{\mathcal{C}}$ rather than the abstract semantic function when computing thresholds. The set of threshold constraints represents an attempt to find useful predicate properties and when widening they help to preserve invariants that might otherwise be lost during widening. Lakhdar et al. [108]

```
new6(A,B) :- B=<99.                    new4(A,B) :- C=1+A,D=1+B,A>=50,new3(C,D).
new5(A,B) :- B>=101.                   new3(A,B) :- A=<99, new4(A,B).
new5(A,B) :- B=<100, new6(A,B).        new3(A,B) :- A>=100, new5(A,B).
new4(A,B) :- C=1+A, A=<49, new3(C,B).  false :- A=0, B=50, new3(A,B).
```

Figure 3.1: The example program `MAP-disj.c.map.pl`

claim that three iterations are enough to capture dependecies between complex loop structures. See [108] for further details. Threshold constraints that are not invariants are simply discarded during widening. We discuss further about the threshold constraints with motivation and example in Chapter 4.

3.4 THE ROLE OF CLP TOOLS IN VERIFICATION

The techniques discussed in the previous section play various roles. The convex polyhedron analysis, together with the helper tool to derive threshold constraints, constructs an approximation of the minimal model of a CHC theory. If *false* (or $false_{ans}$) is not in the approximate model, then the verification problem is solved. Otherwise the problem is not solved; in effect a "don't know" answer is returned. We have found that polyhedron analysis alone is seldom precise enough to solve non-trivial CHC verification problems; in combination with the other tools, it is very effective.

Unfolding can improve the structure of a program, removing some cases of mutual recursion, or propagating constraints upwards towards the integrity constraints, and can improve the precision and performance of convex polyhedron analysis.

Specialisation can remove parts of theories not relevant to the verification problem, and can also propagate constraint downwards from the integrity constraints. Both of these have a beneficial effect on performance and precision of polyhedron analysis.

Analysis of a query-answer program (with respect to *false*) is in effect the search for a derivation tree for *false*. Its effectiveness in CHC verification problems is variable. It can sometimes worsen performance since the query-answer transformed program is larger and contains more recursive dependencies than the original. On the other hand, one seldom loses precision and it is often more effective in allowing constraints to be propagated upwards (through the ans predicates) and downwards (through the $query$ predicates).

3.4.1 *Application of the tools*

We illustrate the tools on a running example (Figure 3.1), one of the benchmark suite of the VeriMAP system [37]. The result of applying unfolding is shown in Figure 3.2 (omitting the definitions of the unfolded predicates `new4`, `new5` and `new6`, which are no longer reachable from *false*). The unfolding strategy we adopt is the following: the predicate dependency graph of a program consists of the set of edges (p, q) such that there is clause where p is the predicate of the head and q is a predicate occurring in the body. We perform a depth-first search of the

```
false :- A=0, B=50, new3(A,B).
new3(A,B) :- A=<99, C = 1+A, A=<49, new3(C,B).
new3(A,B) :- A=<99, C = 1+A, D = 1+B, A>=50, new3(C,D).
new3(A,B) :- A>=100, B>=101.
new3(A,B) :- A>=100, B=<100, B=<99.
```

Figure 3.2: Result of unfolding `MAP-disj.c.map.pl`

```
false_ans :- false_query, A=0, B=50, new3_ans(A,B).
new3_ans(A,B) :- new3_query(A,B), A=<99, C = 1+A, A=<49, new3_ans(C,B).
new3_ans(A,B) :- new3_query(A,B),A=<99,C is 1+A,D is 1+B, A>=50, new3_ans(C,D).
new3_ans(A,B) :- new3_query(A,B), A>=100, B>=101.
new3_ans(A,B) :- new3_query(A,B), A>=100, B=<100, B=<99.
new3_query(A,B) :- false_query, A=0, B=50.
new3_query(A,B) :- new3_query(C,B), C=<99, A = 1+C, C=<49.
new3_query(A,B) :- new3_query(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
false_query.
```

Figure 3.3: The query-answer transformed program for program of Figure 3.2

predicate dependency graph, starting from *false*, and identify the backward edges, namely those edges (p, q) where q is an ancestor of p in the depth-first search. We then unfold every body call whose predicate is not at the end of a backward edge (note that it may cause an exponential blow-up of clauses). In Figure 3.1, we thus unfold calls to `new4`, `new5` and `new6`.

The query-answer transformation is applied to the program in Figure 3.2, with respect to the goal `false` resulting in the program shown in Figure 3.3. The model of the predicate `new3_query` corresponds to those calls to `new3` that are reachable from the call in the integrity constraint. Explicit representation of the query predicates permits more effective propagation of constraints from the integrity clauses during model approximation.

The splitting transformation is now applied to the program in Figure 3.3. We do not show the complete program, which contains 30 clauses. Figure 3.4 shows the split definition of `new3_query`, which is split since the last two clauses for `new3_query` in Figure 3.3 have mutually disjoint constraints, when projected onto the head variables.

```
new3_query___1(A,B) :- false_query___1, A=0, B=50.
new3_query___1(A,B) :- new3_query___1(C,B), C=<99, A = 1+C, C=<49.
new3_query___1(A,B) :- new3_query___2(C,B), C=<99, A = 1+C, C=<49.
new3_query___2(A,B) :- new3_query___1(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
new3_query___2(A,B) :- new3_query___2(C,D), C=<99, A = 1+C, B = 1+D, C>=50.
```

Figure 3.4: Part of the split program for the program in Figure 3.3

```
false_query___1 :- []
new3_query___1(A,B) :- [1*A>=0,-1*A>= -50,1*B=50]
new3_query___2(A,B) :- [1*A>=51,-1*A>= -100,1*A+ -1*B=0]
```

Figure 3.5: The convex polyhedral approximate model for the split program

A convex polyhedron approximation is then computed for the split program, after computing threshold constraints for the predicates. The resulting approximate model is shown in Figure 3.5 as a set of constrained facts. Since the model does not contain any constrained fact for `false_ans` we conclude that `false_ans` is not a consequence of the split program. Hence, applying the various correctness results for the unfolding, query-answer and splitting transformations, `false` is not a consequence of the original program.

DISCUSSION OF THE EXAMPLE. Application of the convex polyhedron tool to the original, or the intermediate programs, does not solve the problem; all the transformations are needed in this case, apart from redundant argument filtering, which only affects efficiency. The ordering of the tool-chain can be varied somewhat, for instance switching query-answer transformation with splitting or unfolding. In our experiments we found the ordering in Figure 3.6 to be the most effective because of the following reason. Predicate splitting allows deriving disjunctive invariants using abstract interpreter as an analyzer, whereas the threshold constraints control the precision of the analyzer. Therefore these two components appear just before the application of CHA. Usually the input program is automatically generated as an output of some tools, the predicates in it may contain arguments which are not necessary to prove the property in question. Therefore we can get the most out of RAF if we apply it in the input program. The component that follows it, that is FU, collects constraints and sometimes removes loop intricacies (for example mutual recursion). This precedes QA, which "compiles" goal directness into the program so that the bottom-up analyzer can be benefitted from it. However if we exchange FU with QA, we may not achieve the same level of simplification from FU, as QA can further complicate the program structure.

The model of the query-answer program is finite for this example. However, the problem is essentially the same if the constants are scaled; for instance we could replace 50 by 5000, 49 by 4999, 100 by 10000 and 101 by 10001, and the problem is essentially unchanged. We noted that some CHC verification tools applied to this example solve the problem, but essentially by enumeration of the finite set of values encountered in the search. Such a solution does not scale well. On the other hand the polyhedral abstraction shown above is not an enumeration; an essentially similar polyhedron abstraction is generated for the scaled version of the example, in the same time. The VeriMAP tool [37] also handles the original and scaled versions of the example in the same time.

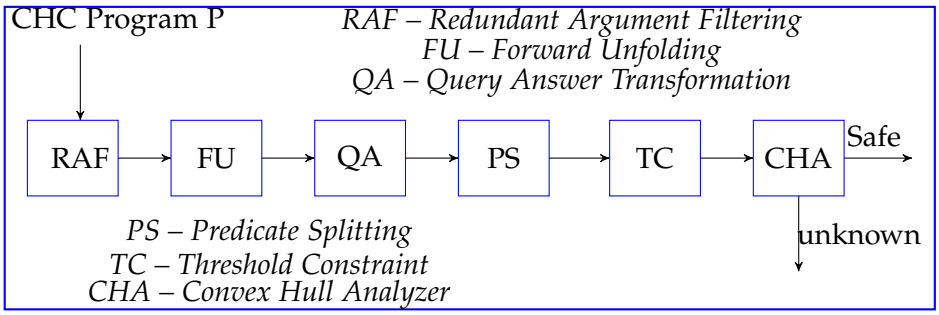

Figure 3.6: *The basic tool chain for CHC verification.*

3.5 COMBINING OFF-THE-SHELF TOOLS: EXPERIMENTS

The motivation for our tool-chain, summarised in Figure 3.6, comes from our example program, which is a simple yet challenging program. We applied the tool-chain to a number of benchmarks from the literature, taken mainly from the repository of Horn clause benchmarks in SMT-LIB2 (https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/) and other sources including [62] and some of the VeriMap benchmarks [37]. We selected these examples because many of them are considered challenging since they cannot be solved by one or more of the state-of-the-art-verification tools discussed below. Programs taken from the SMT-LIB2 repository are first translated to CHC form. We assume that the constraints are over the theory of linear arithmetic and are quantifier free. The results are summarised in Table 3.1.

In Table 3.1, columns Program and Result respectively represent the benchmark program and the results of verification using our tool combination. Problems marked with (*) could not be handled by our tool-chain since they contain numbers which do not fit in 32 bits, the limit of our Ciao Prolog implementation. whereas problems marked with (**) are solvable by simple ad hoc modification of the tool-chain, which we are currently investigating (see Section 3.7). Problems such as systemc-token-ring.01-safeil.c contain complicated loop structure with large strongly connected components in the predicate dependency graph and our convex polyhedron analysis tool is unable to derive the required invariant. However overall results show that our simple tool-chain begins to compete with advanced tools like HSF [64], VeriMAP [37], TRACER [85], *etc.* We do not report timings, though all these results are obtained in a matter of seconds, since our tool-chain is not at all optimised, relying on file input-output and the individual components are often prototypes.

3.6 DISCUSSION AND RELATED WORK

The most similar work to ours is by De Angelis et al. [38] which is also based on CLP program transformation and specialisation. They construct a sequence of transformations of P, say, $P, P_1, P_2, \ldots, P_k$; if P_k contains no clause with head *false* then the verification problem is solved. A proof of unsafety is obtained if P_k contains a clause *false* $\leftarrow$. Both our approach

Table 3.1: Experiments results on CHC benchmark program

SN	Program	Result	SN	Program	Result
1	MAP-disj.c.map.pl	verified	17	MAP-forward.c.map.pl	verified
2	MAP-disj.c.map-scaled.pl	verified	18	tridag.smt2	verified
3	t1.pl	verified	19	qrdcmp.smt2	verified
4	t1-a.pl	verified	20	choldc.smt2	verified
5	t2.pl	verified	21	lop.smt2	verified
6	t3.pl	verified	22	pzextr.smt2	verified
7	t4.pl	verified	23	qrsolv.smt2	verified
8	t5.pl	verified	24	INVGEN-apache-escape-absolute	verified
9	pldi12.pl	verified	25	TRACER-testabs15	verified
10	INVGEN-id-build	verified	26**	amebsa.smt2	verified
11	INVGEN-nested5	verified	27**	DAGGER-barbr.map.c	verified
12	INVGEN-nested6	verified	28*	sshsimpl-s3-srvr-1a-safeil.c	NOT
13	INVGEN-nested8	verified	29	sshsimpl-s3-srvr-1b-safeil.c	NOT
14	INVGEN-svd-some-loop	verified	30*	bandec.smt2	NOT
15	INVGEN-svd1	verified	31	systemc-token-ring.01-safeil.c	NOT
16	INVGEN-svd4	verified	32*	crank.smt2	NOT

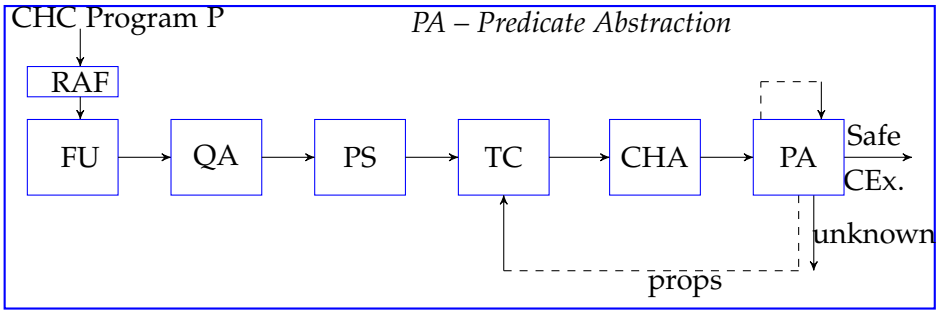

Figure 3.7: *Future extension of our tool-chain.*

and theirs repeatedly apply specialisations preserving the property to be proved. However the difference is that their specialisation techniques are based on unfold-fold transformations, with a sophisticated control procedure controlling unfolding and generalisation. Our specialisations are restricted to redundant argument filtering and the query-answer transformation, which specialises predicate answers with respect to a goal. Their test for success or failure is a simple syntactic check, whereas ours is based on an abstract interpretation to derive an over-approximation. Informally one can say that the hard work in their approach is performed by the specialisation procedure, whereas the hard work in our approach is done by the ab-

stract interpretation. We believe that our tool-chain-based approach gives more insight into the role of each transformation. The weakness of our approach at the moment is that we cannot prove unsafety. This needs either generation of counterexamples or iterative application our procedure and the use of similar idea as in [38] for the proof of unsafety.

Work by Gange et al. [62] is a top-town evaluation of CLP programs which records certain derivations and learns only from failed derivations. This helps to prune further derivations and helps to achieve termination in the presence of infinite executions. Duality[1] and HSF(C) [64] are examples of the CEGAR approach (Counter-Example-Guided Abstraction Refinement). This approach can be viewed as property-based abstract interpretation based on a set of properties that is refined on each iteration. The refinement of the properties is the key problem in CEGAR; an abstract proof of unsafety is used to generate properties (often using interpolation) that prevent that proof from arising again. Thus, abstract counter-examples are successively eliminated. The relatively good performance of our tool-chain, without any refinement step at all, suggests that finding the right invariants is aided by a tool such as the convex polyhedron solver and the pre-processing steps we applied. In Figure 3.7 we sketch possible extensions of our basic tool-chain, incorporating a refinement loop and property-based abstraction.

It should be noted that the query-answer transformation, predicate splitting and unfolding may all cause a blow-up in the program size. The convex polyhedron analysis becomes more effective as a result, but for scalability we need more sophisticated heuristics controlling these transformations, especially unfolding and splitting, as well as lazy or implicit generation of transformed programs, using techniques such as a fixpoint engine that simulates query-answer programs [25].

3.7 CONCLUDING REMARKS AND FUTURE WORK

We have shown that a combination of off-the-shelf tools from CLP transformation and analysis, combined in a sensible way, is surprisingly effective in CHC verification. The component-based approach allowed us to experiment with the tool-chain until we found an effective combination. This experimentation is continuing and we are confident of making improvements by incorporating other standard techniques and by finding better heuristics for applying the tools. Further we would like to investigate the choice of chain suitable for each example since more complicated problems can be handled just by altering the chain. We also suspect from initial experiments that an advanced partial evaluator such as ECCE [116] will play a useful role. Our results give insights for further development of automatic CHC verification tools. We would like to combine our program transformation techniques with abstraction refinement techniques and experiment with the combination.

1 http://research.microsoft.com/en-us/projects/duality/

4

CONSTRAINT SPECIALISATION IN HORN CLAUSE VERIFICATION

With John P. Gallagher

Abstract
We present a method for specialising the constraints in constrained Horn clauses with respect to a goal. We use abstract interpretation to compute a model of a query-answer transformation of a given set of clauses and a goal. The effect is to propagate the constraints from the goal top-down and propagate answer constraints bottom-up. We use the constraints from the model to compute a specialised version of each clause in the program. The specialisation procedure can be repeated to yield further specialisation. The approach is independent of the abstract domain and the constraint theory underlying the clauses. Experimental results on verification problems show that this is an effective transformation, both in our own verification tools (convex polyhedra analyser) and as a pre-processor to other Horn clause verification tools.

Keywords: constraint specialisation, query-answer transformation, Horn clauses, abstract interpretation, convex polyhedral analysis.

4.1 INTRODUCTION

In this chapter, we present a method for specialising the constraints in constrained Horn clauses, CHCs in short (also called constraint logic programs) with respect to a goal. The verification problem that we address has the following setting: a set of constrained Horn clauses formalises some system and the goal is an atomic formula representing a property of that system. We wish to check whether the goal is a consequence of the Horn clauses. The constraint specialisation procedure uses abstract interpretation to compute constraints propagated both from the goal top-down and constraints in the clauses bottom-up. Then we construct a specialised version of each clause by inserting the relevant constraints, without unfolding the clauses at all. As a result, each clause is further strengthened or removed altogether, while preserving the derivability of the goal.

Verification of this specialised set of clauses becomes more effective since some implicit invariants in the original clauses are discovered and made explicit in the specialised version. A central problem in all automatic verification procedures is to find invariants, and this is the underlying reason for the usefulness of our constraint specialisation procedure. The approach is independent of the abstract domain and the constraint theory.

While specialisation has been applied to verification and analysis problems before, the novelty of our procedure is to do specialisation without any unfolding. The only specialisation is to strengthen constraints within each clause, possibly eliminating a clause if its constraints become unsatisfiable. This seems to capture the essence of the role of constraint propagation,

```
int a ,b;
while (*) { // loop invariant l(a,b)
       a = a + b;
       b = b + 1;
}
```

(a) Example program `Loop_add`

$$a = 1 \wedge b = 0 \rightarrow l(a, b)$$
$$l(a', b') \wedge a = a' + b' \wedge b = b' + 1 \rightarrow l(a, b)$$
$$l(a, b) \rightarrow a \geqslant b$$

(b) Verification conditions for `Loop_add`

Figure 4.1: Motivating example

```
c1. l(A,B):- A=1, B=0.
c2. l(A,B):- A=C+D, B=D+1, l(C,D).
c3. false :- B>A, l(A,B).
```

Figure 4.2: Verification conditions for `Loop_add` in CLP syntax

separated from other operations such as clause unfolding. Somewhat surprisingly, this apparently limited form of specialisation is capable of handling a lot of verification benchmarks on its own; on the other hand, due to its simple form, constraint specialisation is a useful preprocessing step for verification tools incorporating a wider range of techniques. We postulate that making invariants explicit contributes positively to the effect of other constraint manipulation operations such as widening and interpolation. We therefore present the procedure as a useful tool in a toolbox for verification of constrained Horn clauses.

MOTIVATING EXAMPLE We present an example program in Figure 4.1. The problem is to show that if $a = 1 \wedge b = 0$ holds before executing the program `Loop_add` in Figure 4.1a (taken from [38]) then $a \geqslant b$ holds after executing it. Figure 4.1b shows the Horn clauses whose satisfiability establishes this property (its equivalent representation in Constraint Logic Programing (CLP) syntax is shown in Figure 4.2). The predicate $l(a, b)$ corresponds to the loop invariant as indicated in Figure 4.1a. The final clause in Figure 4.1b can be equivalently written as $l(a, b) \wedge b > a \rightarrow \mathit{false}$. It is a simple but still challenging problem for many verification tools for constrained Horn clauses. The loop invariant $a \geqslant b \wedge b \geqslant 0$ proves this program safe, and in the CHC version that invariant is sufficient to establish that *false* is not derivable. Finding this invariant is a challenging task for many state of the art verification tools. For example QARMC [65] or SeaHorn [70] (using only the PDR engine) do not terminate on this program. However, SeaHorn (with PDR and the abstract interpreter IKOS) solves it in less than a second. The tool based on specialisation of Horn clauses [38] needs at least forward and backward iteration to solve this problem. We discuss how our constraint specialisation solves this example without needing further processing.

4.1.1 *Related Work*

There is a good deal of related work, since our procedure draws on a number of different techniques which have been applied in different contexts and languages. The basic specialisation and analysis techniques that we apply are well known, though we are not aware of

previous work combining them in the way we did or applying them effectively in verification problems.

CONSTRAINT STRENGTHENING Methods for strengthening the constraints in logic programs go back at least to the work of Marriott *et al.* on most specific logic programs [123]. In that work the constraints were just equalities between terms and the strengthening was goal-independent. We say a clause $H \leftarrow \phi, B_1, \ldots, B_n$ is a strengthened version of $H \leftarrow \psi, B_1, \ldots, B_n$ if $\phi \rightarrow \psi$. This notion can be easily extended to a (logic) program. A program P is a strengthened (more specific) version of another program P' if each clause in P is a strengthened version of the corresponding clause in P'. In [53] the idea was similar but it was extended to strengthen constraints while preserving the answers with respect to a goal. Constraint strengthening was also applied for extracting determinacy in logic program executions [36], in a goal-dependent setting, and arithmetic constraints were also handled. The purpose of constraint strengthening in these works was to improve execution efficiency, for example by detecting failure earlier or to allow more efficient compilation.

Furthermore the idea of constraint propagation leading to strengthening is widespread in constraint-based languages and problem-solving frameworks [124, 141] as well as in partial evaluation and program specialisation [50, 52, 57, 89, 90, 106, 107, 111, 112, 144].

Our methods for strengthening constraints use established techniques and differ from the works mentioned mainly in the intended application, which is verification in our case. The distinctive aspects of our procedure are (a) that constraints are derived from a global analysis of Horn clause derivations starting from a given goal, using abstract interpretation, and (b) that we strengthen using constraints derived from successful derivations of the goal. This means that clause specialisation can eliminate failing or looping derivations, which is not allowed in some of the related work mentioned above, whose aim is to preserve the computational behaviour of a program including failure and looping.

QUERY-ANSWER TRANSFORMATIONS AND RELATED METHODS A central tool in our procedure is query-answer transformation, which is related to the so-called “magic set" transformation. We prefer the terminology query-answer transformation to the name magic set transformation in the literature, as it reflects the purpose of the transformation.

The relevance of the query-answer transformation to the verification problem is that the search for a proof of A from a set of Horn clauses P can be attempted “top-down" starting from the error state (or “query" encoding $\neg A$), or “bottom-up" using answers derived from the clauses of P. A query-answer transformation specialises the clauses in P with respect to the goal A and allows bottom-up and top-down search to be combined.

Combining top-down and bottom-up search is only purpose of query-answer transformation in our procedure, rather than (as in some other variants in the literature mentioned below) to improve efficiency of specific analysis or query procedures. It enables stronger constraints to be derived, namely those that hold in all derivations of a given goal but not necessarily in all models of the clauses. The transformation we use was essentially described in [54].

The “magic set" transformation originated in the field of Datalog query processing in the 1980s [9, 135, 146]. In the Datalog context, the transformation was invented to combine the ef-

ficiency of bottom-up evaluation with focussed top-down search starting from a given query. The transformation with respect to the query returns a set of Datalog rules and facts that are extensions of the original ones except that they contain extra conditions (the so-called "magic predicates") expressing the dependence on the query, together with extra rules for these magic predicates. The resulting rules and facts can be evaluated "bottom-up" allowing efficient algorithms for bottom-up evaluation to be applied, without the potential inefficiency of an undirected goal-independent search. While the "magic sets" and "magic templates" techniques for Datalog also incorporate input-output modes, the query-answer transformation that we use does not require these, including only what is necessary for expressing derivations of a goal with a depth-first left-to-right computation rule.

As with other applications of query-answer transformations in logic program analysis [26, 36, 42, 55], there is also the practical motivation that analysis tools for goal-independent analysis can be reused for goal-dependent analysis. A goal-independent analysis derives an approximation of the model of a program which implicitly expresses the behaviour of all goals, whereas goal-dependent analysis derives an approximation of a top-down derivation of a specific set of goals. Algorithms for goal-independent analysis are generally simpler to implement, but precision is generally greater with a goal-dependent analysis. A query-answer transformation allows the goal-dependence to be "compiled in" to the program clauses; analysis of the resulting transformed clauses using a goal-independent analysis framework yields results that are at least as precise as with a goal-dependent analysis.

Although not formulated as a program transformation, the semantic concept of a *minimal function graph* is related to query-answer transformations. Given a function $f : A \to B$, its function graph is the set of pairs $\{x \mapsto f(x) \mid x \in A\}$. Given some "call" to the function, say $f(a)$, the minimal function graph is the smallest set of pairs $x \mapsto f(x)$ sufficient to evaluate $f(a)$ (ignoring calls to subsidiary functions). This is in general a subset of the function graph of f. For applications in program analysis one is satisfied with some useful superset of the minimal function graph, which might be easier to compute than the precise minimal function graph. Minimal function graph semantics have been formulated for both functional [88, 92] and logic programming [57, 148] languages and applied to program analysis problems. The "query" or "magic" predicates of the transformations appear in minimal function graph constructions as a set of function invocations computed top-down from the call. Informally, a query-answer transformation for logic programs could be viewed as a "compilation" of the minimal graph semantics with respect to a specific program and goal, though further study is needed to formalise this view.

ABSTRACT INTERPRETATION Abstract interpretation [32] is a static program analysis technique which derives sound over-approximations of programs by computing abstract semantics. Abstract interpretation over a domain of convex polyhedra was first achieved by Cousot and Halbwachs [31] and applied to constraint logic programs by Benoy and King [10]. Abstract interpretation over convex polyhedra was incorporated in a program specialisation algorithm by Peralta and Gallagher [130]. The method of widening with thresholds for increasing the precision of widening over the domain of convex polyhedra was first presented by Halbwachs *et al.* [72]. We adapted and applied a technique for generating threshold con-

straints presented by Lakhdar-Chaouch *et al.* [108]. We apply abstract interpretation over the domain of convex polyhedra to derive an over-approximation of the model of the query-answer transformed program.

VERIFICATION BY SPECIALISATION The use of program transformation to verify properties expressed as constraint logic programs was pioneered by Pettorossi and Proietti [132] and Leuschel [113] and continues in recent work by De Angelis *et al.* [38, 37]. Transformations that preserve the minimal model (or other suitable semantics) of logic programs are applied systematically to make properties explicit. Our approach can be regarded as identifying the essence of these tools, namely constraint propagation forwards and backwards within the program. The program specialisation technique supercompilation [144], which also inherently involves constraint propagation through "driving", was applied as a tool to verify statements [104, 105, 118]. Supercompilation is a language-independent concept originally developed for the functional language REFAL and later adapted for some other languages as well.

CLP VERIFICATION TOOLS Verification of CLP programs has been studied for some time. Our aim in this chapter is not to demonstrate a new verification tool but to identify a transformation that can often verify programs on its own and also benefits CLP verification tools generally as a pre-processor. The work closest to ours is by De Angelis et al. [38]; that method also includes forward and backward propagation of constraints using *fold-unfold* transformation. The resulting program in their approach can blow up in size with respect to the original program when specialisation with the "polyvariant" generalisation strategy is used, whereas constraint specialisation cannot. Furthermore the forward propagation method in that work uses a program reversal which can only be applied to linear Horn clauses (a transition system) whereas we can handle also non-linear clauses. However there are various methods for linearisation of certain classes of Horn clauses in the literature [40, 100].

Much other work on CLP verification exists, much of it based on property abstraction and refinement using interpolation, for example [23, 64, 134, 8, 15, 69]. Our specialisation technique is not directly comparable to these methods, but as we have shown in experiments with QARMC and Eldarica, constraint specialisation can be used as a pre-processor to such tools, increasing their effectiveness. The model checking algorithm implemented in Eldarica for Horn clause verification is similar in spirit to the one described in [64] but uses disjunctive interpolation for counterexamples generalisation, which is strictly more general than tree interpolation [136]. We use the approach described in this chapter as a pre-processor of Horn clauses in the tool RAHFT [102].

4.1.2 *Overview and contributions of the chapter*

In Section 4.2 the relevant notation and theory concerning CHCs is presented. Following this, Section 4.3 describes various methods and techniques employed in our procedure, in the form required. These include an algorithm for computing an abstract interpretation of a set of CHCs over the domain of convex polyhedra in Section 4.3.1 and the details of the query-answer transformation in Section 4.3.2. Section 4.4 contains the main contribution, namely the

procedure for constraint specialisation. Section 4.5 puts the procedure in the context of verification, explaining the role of CHC integrity constraints. Section 4.6 contains the experimental evaluation of the procedure. Finally Section 4.7 presents the conclusions.

The contributions of this work are as follows.

- We present a method for specialising the constraints in the clauses using query-answer transformation and abstract interpretation (see Section 4.4).
- We demonstrate the effectiveness of the transformation by applying it to Horn clause verification problems (see Section 4.6).

Experimental results on verification problems show that this is an effective transformation, propagating information both backwards from the statement to be proved, and forwards from the Horn clauses. We show its effectiveness both in our own verification tools and as a pre-processor to other Horn clause verification tools. In particular, we run our specialisation procedure as a pre-processor to our *convex polyhedra analyser*, to the state of the art verification tools like QARMC [64, 134] and Eldarica [80].

4.2 PRELIMINARIES

A constrained Horn clause (CHC) is a first order predicate logic formula of the form $\forall(\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k) \rightarrow p(X))$ $(k \geqslant 0)$, where ϕ is a conjunction of constraints with respect to some constraint theory, X_i, X are (possibly empty) vectors of distinct variables, $p_1, \ldots, p_k, p$ are predicate symbols, $p(X)$ is the head of the clause and $\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k)$ is the body. The arguments of a predicate are always regarded as a tuple; when we write $p(X)$ or $p(a)$, then X and a stand for (possibly empty) tuples of variables and constants respectively.

A set of CHCs can be regarded as a CLP program. Unlike CLP, CHCs are not always regarded as executable programs, but rather as specifications or semantic representations of other formalisms. However the semantic equivalence of CHC and CLP allows techniques developed in one framework to be applied to the other. We follow the syntactic conventions of CLP and write a Horn clause as $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$. In this chapter we take the constraint theory to be linear arithmetic with the relation symbols $\leqslant, \geqslant, <, >$ and $=$, but the contributions of the chapter are independent of the constraint theory.

4.2.1 *Interpretations and models*

An interpretation of a set of CHCs is a truth assignment to each atomic formula $p(a)$, where p is a predicate and a is a tuple of constants from the constraint theory. An interpretation is represented as a set of *constrained facts* of the form $A \leftarrow \phi$ where A is an atomic formula $p(Z)$ where Z is a tuple of distinct variables and ϕ is a constraint over Z. The constrained fact $A \leftarrow \phi$ stands for the set of ground facts $A\theta$ (where θ is a grounding substitution) such that $\phi\theta$ holds in the constraint theory. For example the constrained fact $p(X) \leftarrow X \geqslant 0$ represents the infinite set $\{p(0), p(1), p(2), \ldots\}$. An interpretation M is the set of all ground facts denoted by its elements. $M_1 \subseteq M_2$ if the set of denoted ground facts of M_1 is contained

in the set of denoted ground facts of M_2. An interpretation M satisfies a CHC $p_0(X_0) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$, if M contains constrained facts $\{p_i(X_i) \leftarrow \phi_i \mid 0 \leqslant i \leqslant n\}$, and $\forall(\phi_0 \leftarrow (\phi \wedge \bigwedge_{i=1}^{n} \phi_i))$ is true. In other words, the set of ground facts dentoed by $p_0(X_0) \leftarrow (\phi \wedge \bigwedge_{i=1}^{n} \phi_i)$ is subset of the set of ground facts dentoed by $p_i(X_0) \leftarrow \phi_0$.

MINIMAL MODELS A model of a set of CHCs is an interpretation that satisfies each clause. A set of CHCs P has a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$. Let S_P^D be the immediate consequences operator, an extension of the standard T_P operator from logic programming, extended to handle the constraint domain D [81, Section 4]. $M[\![P]\!]$, which is equal to the least fixed point of S_P^D, can be computed as the limit of the sequence of interpretations $\emptyset, S_P^D(\emptyset), S_P^D(S_P^D(\emptyset)), \ldots$. The abstract interpretation of CHC clauses presented in Section 4.3.1, uses this sequence as the basis of the model computation. From now on, whenever we talk about a model of Horn clauses, we refer to its minimal model.

Given two constraints ϕ and ψ over some constraint theory T, we say ϕ is stronger than ψ if $T \models \forall(\phi \rightarrow \psi)$.

4.2.2 *Integrity constraints and safety*

A CHC of the form *false* $\leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$ is called an integrity constraint. The predicate *false* is false in all interpretations. We use integrity constraints to capture safety properties, as discussed later in Section 4.5.2.

Definition 4.1 (Safety). *A set of Horn clauses is safe (unsafe) if and only if it has a model (no model).*

The body $\phi, p_1(X_1), \ldots, p_k(X_k)$ of an integrity constraint represents an unsafe condition. It follows from Definition 4.1 that a set of CHCs is safe if and only if $\phi, p_1(X_1), \ldots, p_k(X_k)$ is not satisfiable for each integrity constraint *false* $\leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$.

4.3 METHODS AND TECHNIQUES

This section describes the techniques used in the constraint specialisation procedure, including abstract interpretation over convex polyhedra and the query-answer transformation. These are methods drawn from the literature; the contribution of the section is to present them in a form suitable for integration in the procedure, and present an efficient implementation of the abstract interpretation of CHCs.

4.3.1 *Abstract Interpretation over the domain of convex polyhedra*

Convex polyhedral analysis (CPA) [31] is a program analysis technique based on abstract interpretation [32]. When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P, where M' contains at most one constrained fact $p(X) \leftarrow \phi$ for each predicate p. The constraint ϕ is a conjunction of linear inequalities, representing a convex polyhedron. The first application of CPA to CHCs was by Benoy and King [10]. In

this section we develop an algorithm for CPA of CHCs incorporating a number of features enhancing precision and efficiency.

We summarise briefly the elements of convex polyhedral analysis for CHC; further details can be found in [31, 10]. Let $\mathcal{A}^k$ be the set of convex polyhedra of dimension k. Let P be a set of CHCs containing n predicates, say $p_1, \ldots, p_n$, where the arity of p_i is $ar(p_i)$. The *abstract domain* $\mathcal{D}_P$ for P (or just $\mathcal{D}$ when P is clear from context) is the set of n-tuples of convex polyhedra of the respective dimension, that is $\mathcal{D} = \mathcal{A}^{ar(p_1)} \times \cdots \times \mathcal{A}^{ar(p_n)}$. Let the empty polyhedron of dimension k be denoted $\bot_k$ (or just $\bot$ when the dimension is clear from context). Inclusion of polyhedra is a partial order on $\mathcal{A}^k$ and the partial order $\sqsubseteq$ on $\mathcal{D}$ is its point-wise extension. The convex hull of two polyhedra $d_1, d_2 \in \mathcal{A}^k$ is denoted $d_1 \sqcup d_2$, and the least upper bound $\sqcup$ of tuples in $\mathcal{D}_P$, say $\langle d_1, \ldots, d_n \rangle$ and $\langle e_1, \ldots, e_n \rangle$, is $\langle d_1 \sqcup e_1, \ldots, d_n \sqcup e_n \rangle$. Given an element $\langle d_1, \ldots, d_n \rangle \in \mathcal{D}_P$, define the *concretisation* function γ such that $\gamma(\langle d_1, \ldots, d_n \rangle) = \{\langle p_1(a_1), \ldots, p_n(a_n) \rangle \mid a_i \text{ is a point in } d_i, 1 \leqslant i \leqslant n\}$. Let an *abstract semantic function* be $F_P : \mathcal{D}_P \to \mathcal{D}_P$ satisfying the condition $S_P^D \circ \gamma \subseteq \gamma \circ F_P$, where F_P is monotonic with respect to $\sqsubseteq$ and S_P^D is the immediate consequences operator mentioned in Section 4.2. Let the increasing sequence $Y_0, Y_1, \ldots$ be defined as follows. $Y_0 = \bot, Y_{n+1} = F_P(Y_n)$. These conditions are sufficient to establish that the limit of the sequence, say Y, exists and satisfies $\gamma(Y) \supseteq lfp(S_P^D) = M[\![P]\!]$ [30].

Since $\mathcal{D}_P$ contains infinite increasing chains, the sequence can be infinite. The use of a *widening* operator for convex polyhedra is needed to ensure convergence of the abstract interpretation. Define the sequence $Z_0 = Y_0$, $Z_{n+1} = Z_n \nabla F_P(Z_n)$ where ∇ is a widening operator for convex polyhedra [31]. The conditions on ∇ ensure that the sequence stabilises; thus for some finite j, $Z_i = Z_j$ for all $i > j$ and furthermore Z_j is an upper bound for the sequence $\{Y_i\}$. The value Z_j thus represents, via the concretisation function γ, an over-approximation of the least model of P. Furthermore much research has been done on improving the precision of widening operators, for example, widening-upto, or widening with thresholds [72, 73]. The widening upto operator ($\triangledown_T$) for convex polyhedra with respect to a set T of constraints (the threshold) is a widening operator $Z_1 \triangledown_T Z_2$ such that for all $\phi \in T$, $Z_1 \to \phi \wedge Z_2 \to \phi$ implies that $Z_1 \triangledown_T Z_2 \to \phi$. In other words the widening-upto operator preserves as many of the constraints in the threshold as possible.

4.3.1.1 *Algorithm for convex polyhedral approximation of CHCs*

Given the elements of convex polyhedral analysis summarised above, we present the algorithm for computing a polyhedral approximation of a set of CHCs. A naive algorithm to compute the limit of the sequence $Z_0, Z_1, Z_2, \ldots$ is given in Algorithm 4.1. This naive algorithm is just a stepping stone to present the main algorithm in Figure 4.2. Given a clause $p(X) \leftarrow Body$, the function call $solve(p(X), Body, Z_i)$ returns a constrained fact $p(X) \leftarrow \phi$, where ϕ is the result of solving $Body$ in the current approximation Z_i (note that ϕ is a constraint in the theory of linear arithmetic). More precisely, if $Body = \psi, p_1(X_1), \ldots, p_r(X_r)$ then $\phi = (\psi \wedge \phi_1 \wedge \ldots \wedge \phi_r)|_X$, where $p_i(X_i) \leftarrow \phi_i$ (for $i = 1 \ldots k$) is a (renamed) constrained fact in Z_i. We assume that the constraint theory admits a projection operator, and we write $\phi|_X$

Algorithm 4.1: Naive Algorithm for Convex Polyhedral Analysis

Input: A set of CHCs P
Output: *over-approximation of the minimal model of* P

```
1  i ← 0 ;
2  Z_0 ← ⊥ ;
3  New ← ⊥ ;
4  Changed ← {p | p is a predicate in P} ;
5  while Changed ≠ ∅ do
6      foreach (p(X) ← Body) ∈ P do
7          if Body has changed in Changed then
8              New ← New ⊔ solve(p(X), Body, Z_i)
9      Z_{i+1} ← Z_i ∇ (New ⊔ Z_i) ;          /* Upper bound and widen */
10     Changed ← {p | p has changed in Z_{i+1}} ;
11     i ← i + 1
12 return Z_i
```

to mean the projection of ϕ onto the variables X. That is, if Y is the set of variables in ϕ and $Z = Y \setminus X$ then the variables in Z do not occur in $\phi|_X$ and $\phi|_X \equiv \exists Z.\phi$.

Our algorithm, shown in Algorithm 4.2, incorporates generic optimisations for computing fixed points using an ascending chain. We present it in some detail since we are not aware of implementations that incorporate the same range of optimisations and precision enhancements, although all are drawn from the literature. The first step is to compute the strongly connected components (SCCs) of the predicate dependency graph of the set of CHCs. Each component is a set of (non-constraint) predicates; a group is either non-recursive (in which case it is a singleton) or a set of mutually recursive predicates. The algorithm for computing SCCs returns the components in topologically sorted order $C_1, \ldots, C_m$, such that for each C_j, no predicate in C_j depends on any predicate in C_k where $k > j$ [143].

The algorithm proceeds to solve the components in order. A fixed point is computed for each SCC separately. A standard optimisation for recursive SCCs (the semi-naive optimisation) [145] is to keep track of which predicates have a new solution in each iteration. The set *Changed* records the predicates whose solution is changed. This optimisation allows a clause to be ignored on an iteration, if no predicate in its body has changed since the previous iteration. Obviously such a clause can contribute nothing new to the approximation. A recursive SCC is solved when the set *Changed* is empty after some iteration. For non-recursive SCCs, no iteration is needed. The bodies of the clauses for the predicate in that SCC are solved with respect to the current approximation and their solutions are added to the current approximation.

We apply a widening-upto operator ∇_T where T contains a set of threshold constraints computed at the start of the algorithm (Algorithm 4.2, line 2), which we define in the next paragraph. T consists of facts that represent "guesses" for invariants for each predicate p. Any set T does not alter the soundness result of the Algorithm 4.2 (that is, it produces an

Algorithm 4.2: Algorithm for Convex Polyhedral Analysis (CPA)

Input: A set of CHCs P
Output: *over-approximation of the minimal model of* P

$C_1, \ldots, C_m \leftarrow$ SCCs for P ;
$T \leftarrow \mathsf{thresholds}(P)$;
$i \leftarrow 0$;
$Z_0 \leftarrow \bot$;
for $j = 1$ to m **do**
  if C_j *is recursive* **then**
    Changed $\leftarrow \bigcup_{l=1..j} C_l$
    while *Changed* $\neq \emptyset$ **do**
      $New \leftarrow \bot$;
      foreach $(p(X) \leftarrow Body) \in P$ *where* $p \in C_j$ **do**
        if $Body$ *has changed in* $Changed$ **then**
          $New \leftarrow New \sqcup \mathsf{solve}(p(X), Body, Z_i)$
      $Z_{i+1} \leftarrow Z_i \nabla_T (New \sqcup Z_i)$; /* Widen upto */
      Changed $\leftarrow \{p \mid p$ has changed in $Z_{i+1}\}$;
      $i \leftarrow i + 1$
  else
    $New \leftarrow \bot$;
    foreach $(p(X) \leftarrow Body) \in P$ *where* $p \in C_j$ **do**
      $New \leftarrow New \sqcup \mathsf{solve}(p(X), Body, Z_i)$
    $Z_{i+1} \leftarrow Z_i \sqcup New$; /* No widening */
    $i \leftarrow i + 1$
return Z_i

over-approximation of the minimal model of P), but a good choice of thresholds can make a significant difference to the precision of the final result. In our implementation we adapt a method presented by Lakhdar-Chaouch *et al.* [108]. In brief, the method collects constraints by iterating the abstract semantic function F_P three times starting from the "top" ($\top$) element of $\mathcal{D}$, that is, the interpretation which assigns the universal polyhedron (the polyhedron representing the whole space of a given dimension or true constraint) to each predicate. The choice of three iterations is motivated by Lakhdar-Chaouch *et al.*; however, we believe that further experimentation with choices of thresholds would be fruitful.

We define the operation $\mathsf{thresholds}(P)$ as follows. First define a function which splits a constrained fact into a set of constrained facts having a single constraint. $\mathsf{atomconstraints}(p(Z) \leftarrow \phi)$ returns the set of constrained facts $\{p(Z) \leftarrow \phi_i \mid \phi = \phi_1 \wedge \ldots \wedge \phi_k,\ i = 1 \ldots k\}$ where ϕ_i are atomic constraints. The function is extended to apply to sets of constrained facts.

$$\mathsf{atomconstraints}(I) = \bigcup_{p(Z) \leftarrow \phi \in I} \{\mathsf{atomconstraints}(p(Z) \leftarrow \phi)\}.$$

Then define the thresholds function as follows.

$$\mathsf{thresholds}(P) = \mathsf{atomconstraints}(F_P^{(3)}(\top))$$

Following this definition, the threshold constraints generated for our example program in Figure 4.2 is shown in Example 4.1.

Example 4.1 (Threshold Constraints)**.**

```
l(A,B) :- A=1.  l(A,B) :- B=0.  l(A,B) :- B=1.
l(A,B) :- A=2.  l(A,B) :- B=2.
false :- true.
```

4.3.2 *The query-answer transformation*

In Section 4.1.1 we discussed the origins and motivation of the query-answer transformation. In the following, we define it formally. We assume that, for each atom $A = p(t)$ (where t is any arbitrary term), A^a and A^q represent the atoms $p^a(t)$ and $p^q(t)$ respectively.

Definition 4.2 (Query-answer program)**.** *Given a set of CHCs P and an atom A, the (left-) query-answer clauses for P with respect to A, denoted P_A^{qa} or just P^{qa}, are as follows.*

- *(Answer clauses). For each clause $H \leftarrow \phi, B_1, \ldots, B_n$ $(n \geqslant 0)$ in P, P^{qa} contains the clause $H^a \leftarrow \phi, H^q, B_1^a, \ldots, B_n^a$.*
- *(Query clauses). For each clause $H \leftarrow \phi, B_1, \ldots, B_i, \ldots, B_n$ $(n \geqslant 0)$ in P, P^{qa} contains the following clauses:*

$$B_1^q \leftarrow \phi, H^q.$$
$$\ldots$$
$$B_i^q \leftarrow \phi, H^q, B_1^a, \ldots, B_{i-1}^a.$$
$$\ldots$$
$$B_n^q \leftarrow \phi, H^q, B_1^a, \ldots, B_{n-1}^a.$$

- *(Goal clause).* $A^q \leftarrow$ *true.*

The clauses in P^{qa} encode a left-to-right, depth-first computation of the query $\leftarrow A$ for CHC clauses P (that is, the standard CLP computation rule, SLD extended with constraints). This is a complete proof procedure (produces a proof for each provable atom), assuming that all clauses matching a given call are explored in parallel. (Note: the incompleteness of standard CLP proof procedures arises due to the fact that clauses are tried in a fixed order). We can also define a query-answer transformation which encodes atoms in a right-to-left fashion. Since P^{qa} above encodes atoms in a left-to-right fashion, we call such a transformation (left-) query answer transformation for clarity.

The answer clauses arise since there is an answer for the head predicate H if it was queried and all the body atoms have answers and ϕ holds. The query clauses arise since given a clause $H \leftarrow \phi, B_1, \ldots, B_n$ ($n \geqslant 0$), the i^{th} body atom B_i can only be queried if the head H is queried, ϕ holds and all the body atoms up to $i-1$ have answers (in the left-right computation). Finally, the goal clause asserts that the goal A is queried.

The size of query-answer program is quadratic with respect to the size of the original program. This is because we generate n query-answer clauses for each clause in the original program with n non-constraint atoms. So if we have m clauses in the original program and the maximum number of non-constraint atom in any clause is n, then the query-answer program contains at most $n * m + 1$ clauses.

Example 4.2 (Query-answer transformation)**.** *For a given predicate* p*, we represent* p^a *and* p^q *by* `p_a` *and* `p_q` *respectively in textual form. Given the program in Figure 4.2, its query-answer transformation following the Definition 4.2 is shown below. Note that the identifier preceding each clause shows the identifier of the original clause from where it is derived.*

```
%answer clauses
c1. l_a(A,B) :- l_q(A,B), A=1, B=0.
c2. l_a(A,B) :- l_q(A,B), A=C+D, B=D+1, l_a(C,D).
c3. false_a :- false_q, B>A, l_a(A,B).
%query clauses
c2. l_q(A,B) :- l_q(C,D), C=A+B, D=B+1.
c3. l_q(A,B) :- false_q, B>A.
%goal clause
false_q :-  true.
```

Query-answer clauses capture the mutual dependencies of top-down and bottom-up evaluation, since the queries and answers are defined in a single set of clauses. For example, given a clause $p \leftarrow q, p$, the call to p in the body depends on the answers for q (in a top-down left-right evaluation). However the answers for q depend on the calls to p in the head, since q is called from p. Top-down or bottom-up evaluation in isolation would not capture such mutual dependencies between calls and answers.

The relationship of the model of the clauses P^{qa} to the computation of the goal $\leftarrow A$ in P is expressed by the following property[1]. An SLD-derivation in CLP is a sequence $G_0, G_1, \ldots, G_k$ where each G_i is a goal $\leftarrow \phi, B_1, \ldots, B_m$, where ϕ is a constraint and $B_1, \ldots, B_m$ are atoms. In a left-to-right computation, G_{i+1} is obtained by resolving B_1 with a program clause. The model of P^{qa} captures (approximates) the set of atoms that are "called" or "queried" during the derivation, together with the answers (if any) for those calls. This is expressed precisely by Property 4.1.

Property 4.1 (Correctness of query-answer transformation)**.** *Let P be a set of CHCs and A be an atom. Let P^{qa} be the query-answer program for P with respect to A. Then*

(i) if there is an SLD-derivation $G_0, \ldots, G_i$ where $G_0 = \leftarrow A$ and $G_i = \leftarrow \phi, B_1, \ldots, B_m$, then $P^{qa} \models \forall(B_1^q \leftarrow \phi|_{\mathsf{vars}(B_1)})$;

(ii) if there is an SLD-derivation $G_0, \ldots, G_i$ where $G_0 = \leftarrow A$, containing a sub-derivation $G_{j_1}, \ldots, G_{j_k}$, where $G_{j_i} \leftarrow \phi', B, \mathcal{B}'$ and $G_{j_k} = \leftarrow \phi, \mathcal{B}'$, then $P^{qa} \models \forall(B^a \leftarrow \phi|_{\mathsf{vars}(B)})$. (This means that the atom B in G_{j_i} was successfully answered, with answer constraint $\phi|_{\mathsf{vars}(B)}$, where $\mathcal{B}'$ is a conjunction of atoms).

(iii) As a special case of (ii), if there is a successful derivation of the goal $\leftarrow A$ with answer constraint ϕ then $P^{qa} \models \forall(A^a \leftarrow \phi)$.

The correctness of query-answer transformation has already been established by several authors in the logic programming literature, for example, Nilsson [128] and Debray et al. [42]. Note that Property 4.1 allows for the fact that P^{qa} might include some "extra" answers compared with the original program. However, since our proof method using abstract interpretation also makes safe approximations this is not an significant issue. A proofs of unsafety using abstract interpretation (such as those indicated in the table of experimental results in Section 4.6) depend in any case on checking that the "possibly unsafe" result derived from an approximation is actually a concrete counterexample in the original program.

4.4 CONSTRAINT SPECIALISATION

We next present a procedure for specialising CHCs. In contrast to classical specialisation techniques based on partial evaluation with respect to a goal, the specialisation does not

1 Note that the model of P^{qa} might not correspond exactly to the calls and answers in the SLD-computation, since the CLP computation treats constraints as syntactic entities through decision procedures and the actual constraints could differ. Though the form of constraints may differ, the model of a predicate contains exactly the same ground atoms. Therefore this lack of correspondence is not important.

unfold the clauses at all; rather, it computes a specialised version of each clause, in which the constraints from the goal are propagated top-down and answers are propagated bottom-up.

We first make precise what is meant by "specialisation" for CHCs. Let P be a set of CHCs and let A be an atomic formula. The specialisation of P with respect to A is a set of clauses P_A such that for every constraint ϕ over the variables of A, $P \models \forall(A \leftarrow \phi)$ if and only if $P_A \models \forall(A \leftarrow \phi)$. This is a very general definition that allows for many transformations. In practice we are interested in specialisations that eliminate logical consequences of P that have no relevance to A. In this chapter, the word "specialisation" refers just to transformations that strengthen the constraint of each clause, while preserving the general property of specialisation given above.

For each clause $H \leftarrow \mathcal{B}$ in P, P_A contains a new clause $H \leftarrow \phi, \mathcal{B}$ where ϕ is a constraint. If the addition of ϕ makes the clause body unsatisfiable, it is the same as removing the clause, though removal is not essential to the procedure. Clearly P_A may have fewer logical consequences than P but our procedure guarantees that it preserves the logical consequences of P with respect to the (ground instances of) A.

Algorithm 4.3: Algorithm for Constraint Specialisation **(CS)**

Input: A set of CHCs P and an Atom A
Output: A set of CHCs P_s

1 $P^{qa} \leftarrow$ **Query-answer-transformation(** P, A **)** ; /* Definition 4.2 */
2 $M \leftarrow$ **CPA(** P^{qa} **)** ; /* Algorithm 4.2 */
3 $P_s \leftarrow$ **Strengthen-constraints(**P,M**)** ; /* Definition 4.3 */
4 **return** P_s

The algorithm for constraint specialisation is shown in Algorithm 4.3. The inputs are a set of CHCs P and an atomic formula A and the output is a set of specialised clauses. Firstly, it computes a *query-answer transformation* of P with respect to A, denoted P^{qa}, containing predicates p^q and p^a for each predicate p in P (line 1, Algorithm 4.3). Secondly, it computes an over-approximation M of the model of P^{qa} (line 2, Algorithm 4.3). Finally, it strengthens the constraints in the clauses in P by adding constraints from the answer predicates in M producing P_s (line 3, Algorithm 4.3). Next we will explain each step in detail.

4.4.1 *The query-answer transformation*

This was presented in Section 4.3.2. We perform a query-answer transformation of P with respect to the goal *false*. We call the result P^{qa}. It follows from Property 4.1(iii) that if *false* is derivable from P then false^a is derivable from P^{qa}.

4.4.2 *Over-approximation of the model of* P^{qa}

Abstract interpretation of P^{qa} yields an over-approximation of $M[\![P^{qa}]\!]$, say M, containing constrained facts for the query and answer predicates. These represent the calls and answers

generated during all derivations starting from the goal A. In our experiments we use a convex polyhedral approximation (CPA) of $M[\![P^{qa}]\!]$, as described in Section 4.3.1. Using CPA, we derive the following constrained facts for the program in Example 4.2.

Example 4.3 (Over-approximation of the model of the program in Example 4.2)**.**

```
false_q :- true.
l_(A,B) :- true.
l_a(A,B) :- A>=1, A-B>=0, B>=0.
```

For all predicates in a program for which the model contains no constrained fact, we assume that there is a constrained fact for that predicate whose right hand side contains an unsatisfiable constraint.

4.4.3 *Strengthening the constraints in* P

We use the information in the model of P^{qa}, say M, to specialise the original clauses in P. Suppose M contains constrained facts $p^q(X) \leftarrow \phi^q$ and $p^a(X) \leftarrow \phi^a$. (If there is no constrained fact $p^*(X) \leftarrow \phi$ for some p^* then we consider M to contain $p^*(X) \leftarrow \mathit{false}$, as mentioned above).

Given such a set M, define γ_M to be the mapping from atoms to constraints such that $\gamma_M(p^*(X)) = \phi$ for each constrained fact $p^*(X) \leftarrow \phi$, where $*$ is a or q.

Definition 4.3 (Strengthened clauses P_A from a model.)**.** *Let* P *be a set of CHCs,* A *be a goal and* P^{qa} *be the query-answer transformation of* P *with respect to* A*. Let* M *be a model of* P^{qa} *defined by a set of constrained facts. Then* P_A *contains the following clauses:*

$$P_A = \{p(X) \leftarrow \phi, \phi_0, \phi_1, \ldots, \phi_n, p_1(X_1), \ldots, p_k(X_k) \mid \begin{array}{l} p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k) \in P, \\ \phi_0 = \gamma_M(p^a(X)), \phi_i = \gamma_M(p_i^a(X_i)), \\ \mathsf{SAT}(\phi \wedge \phi_0 \wedge \bigwedge_{i=1}^{n} \phi_i)\ \}\end{array}$$

The clauses whose body constraints are unsatisfiable are removed from P_A, since they cannot contribute to feasible derivations (a Horn clause derivation tree whose constraints are unsatisfiable) and do not contribute to the minimal model of P_A. Here we assume that there is exactly one constrained fact in M for each predicate $p^a, p_1^a, \ldots, p_n^a$. Due to the choice of domain for abstract interpretation, we get one constrained fact (a convex polyhedron) for each predicate in the program. Using a richer domain such as the power set of convex polyhedra, we could obtain disjunctive constraints, which could be eliminated from the specialised clauses by program transformation. For example, the clause $p(X) \leftarrow (X > Y \vee Y < X), q(Y)$ can be transformed into $p(X) \leftarrow X > Y, q(Y)$ and $p(X) \leftarrow Y < X, q(Y)$. However, that could cause blow-up in the number of clauses generated.

Note that wherever M contains constrained facts $p^a(X) \leftarrow \phi^a$ and $p^q(X) \leftarrow \phi^q$, we have $\phi^a \rightarrow \phi^q$ since the answers for p are always stronger than the calls to p. Thus it suffices to add only the answer constraints to the clauses in P and we can ignore the model of the query predicates. A special case of this is where M contains a constrained fact $p^q(X) \leftarrow \phi^q$ but there is no constrained fact for $p^a(X)$, or in other words M contains the constrained fact

$p^a(X) \leftarrow$ *false* (meaning that no ground atom exists for the predicate p^a in M). This means that all derivations for $p(X)$ fail or loop in P and so adding the answer constraint *false* for p eliminates looping derivations for p.

Example 4.4 (Constraint specialisation)**.** *Using the model of the query-answer transformed program presented in Example 4.3, the program in Figure 4.2 can be strengthened as follows:*

```
c1.l(A,B):- A=1, B=0, A>=1, A-B>=0, B>=0.
c2.l(A,B):- A=C+D, B=D+1, A>=1, A-B>=0, B>=0, C>=1, C-D>=0, D>=0, l(C,D).
c3.false:- B>A, A>=1, A-B>=0, B>=0, l(A,B).
```

The constraints in the clauses are strengthened by the addition of extra constraints from the model, which are underlined. It can be seen that the constraints in the body of integrity constraint (`c3`*) are unsatisfiable, and thus* `c3` *can be eliminated.*

Specialisation by strengthening the constraints preserves the answers of the goal with respect to which the query-answer transformation was performed. In particular, we have the following property.

Property 4.2 (Soundness of constraint specialisation)**.** *If* P *is a set of CHCs and* P_A *is the set of clauses obtained by strengthening the clause constraints as just described, then* $P \models (A \leftarrow \phi)$ *if and only if* $P_A \models (A \leftarrow \phi)$.

Proof. The proof follows from the standard Theorems (Theorem 6.0.1, Part 4 and Theorem 6.0.1, Part 2 of [82]) and Lemmas (Lemma 3.1 of [123]) from the literature in constraint logic programming.

□

The specialisation and analysis are separate in our approach. More complex algorithms intertwining them can be envisaged, though the benefits are not clear. Our constraint specialisation algorithm can be iterated as depicted in Algorithm 4.4 to obtain further specialsation. Unlike [38] our specialisation is itself a fixpoint computation and combines forward and backward propagation.

Iteration can give further specialisation (as our experiments confirm) since the query-answer transformation encodes a left-to-right top-down query evaluation and thus does not directly propagate constraints from right to left in clause bodies. Hence it can take more than one iteration for answers to propagate from right to left in a clause body. We explain this with the following example.

```
false :- q(X).
q(X) :- p1(X), p2(X).

p1(X) :- X=1.
p1(X) :- X=3.
p2(X) :- X=2.
```

This requires two iterations to show safety. The convex polyhedron approximating `p1` loses information and includes `X=2`. But once the answer for `p2`, that is, `X=2` is propagated after the

first iteration, `p1` fails. However reversing the body atoms clause defining `q(X)` gives safety in one iteration, since the answer `X=2` is propagated to the query for `p1` left-to-right.

Algorithm 4.4: Algorithm for Iterated Constraint Specialisation **(Iterated CS)**

Input: A set of CHCs P and an Atom A
Output: A set of CHCs P_s

```
1 repeat
2 |   P_s ← CS(P) ;                                  /* Algorithm 4.3 */
3 |   swap(P, P_s)
4 until (P = P_s);
5 return P_s
```

COMPLEXITY There are various components in Algorithm 4.3 whose complexity affects its worst-case performance. The number of iterations of the fixpoint computation of an abstract interpretation (the **while** loop in Algorithm 4.2) is bounded by the height of the abstract domain but this is infinite in the case of the domain of convex polyhedra. The widening operation does give a bound but we are not aware of any analysis of the complexity of typical widening operators such as those built in to the Parma Polyhedra Library. Computing the convex hull of polyhedra using Fourier-Motzkin variable elimination has double exponential (in the number of dimensions) worst case time. So it is clear that the worst case using the domain of convex polyhedra can give scalability problems; the practical question is then how it performs in average cases and what to do to trade off precision with complexity. As with all abstract interpretations, both the domain and the widening can be tuned to trade off precision with complexity. A simpler class of polyhedra (for example intervals or octagons) could be used, and widening could be coarsened to ensure faster convergence. Furthermore, we have successfully specialised constraints in clauses with hundreds of variables, and experimental results seem to show that the use of convex polyhedra is feasible when the derived polyhedra are characterised by linear constraints having a low number of variables, avoiding a serious blow-up in the Fourier-Motzkin algorithm. Finally, as mentioned in Section 4.3.2 the query-answer transformation causes a quadratic increase compared to the size of the original clauses.

4.5 APPLICATION TO THE CHC VERIFICATION PROBLEM

In this section we will briefly discuss the origin of Horn clauses in verification problems and then discuss the application of our constraint specialisation in Horn clause verification.

4.5.1 *Origin of Horn clauses in program verification problems*

CHCs provide a logical formalism suitable for expressing the semantics of a variety of programming languages (imperative, functional, concurrent, *etc.*) and computational models

(state machines, transition systems, big- and small-step operational semantics, Petri nets, *etc.*). Program verification usually refers to verification of source programs rather than some intermediate semantic form. Instead of devising verification procedure for each source language (which is a difficult process) we devise a verification procedure for CHCs and then translate source languages to it, saving time and effort. The literature on program analysis and verification contains several methods for generating CHCs from an imperative program, including assertions to be verified, which fall into two broad categories.

1. *Specialising an interpreter*: The translation is usually obtained through specialisation of an interpreter (equivalently partial evaluation). Let P_{imp} be an imperative program written in language L and I be an interpreter of L written in some language (as Horn clauses in our case). Partial evaluation or specialisation of I with respect to P_{imp} produces a specialised interpreter I_s for P_{imp}. I_s can be regarded as the translation of P_{imp} to the language in which the interpreter is written which preserves the semantics of P_{imp}. This approach is taken by Peralta *et al.* [129] and De Angelis, Fioravanti, Pettorossi and Proietti [39].

2. *Hoare style proof rules*: The translation is obtained by applying the proof rules to obtain logical proof subgoals whose satisfiability implies correctness of the original program. This approach is taken by Gurfinkel *et al.* [70], Grebenshchikov, Lopes, Popeea and Rybalchenko [65] and McMillan and Rybalchenko [126].

In both of these, the program semantics can be *small-step*, *big-step* or *mixed*. In the first category this is specified by an interpreter whereas in the second case it is specified by proof rules. The outcome of the translation in both cases is a set of Horn clauses often called *verification conditions* in the literature [39, 126]. There are also other ad-hoc techniques for translation, for example, [43, 83].

The assertions to be verified are manifested in the CHCs as integrity constraints (Section 4.2.2). An assertion ϕ at a given program point in an imperative program is intended as a *safety condition*, namely that whenever control reaches that point, ϕ holds. If $\neg\phi$ can hold at that point, the program is considered unsafe. This is encoded as an integrity constraint of the form $\mathit{false} \leftarrow \neg\phi, \mathcal{B}$, where $\mathcal{B}$ is some formula representing the state at the program point at which ϕ should hold.

We illustrate this via the example program already introduced in Section 4.1 in Figure 4.1, which is taken from [38]. Suppose we would like to prove the Hoare triple $\{a = 1, b = 0\}$ `Loop_add` $\{a \geqslant b\}$. This means starting from a state satisfying $\{a = 1, b = 0\}$, if we execute `Loop_add` and if it terminates then the resulting state satisfies $\{a \geqslant b\}$. Let `l(a,b)` be an unknown loop invariant for the *while loop* in the program `Loop_add`, the above triple holds if the verification conditions in Figure 4.1b are satisfiable, that is, there exists an interpretation that satisfies each clause.

Using CLP syntax the verification conditions can be written as shown in Figure 4.2. Each clause in this example is assigned an identifier (for example `c1` to `c3`) in order to refer them later. The clause `c3` is an integrity constraint expressing the fact that $b > a$ does not hold at the program exit point.

4.5.2 *CHC verification*

In this section, we discuss the application of our constraint specialisation in Horn clause verification. As discussed in Section 4.5.1, assertions representing safety properties are translated into integrity constraints, clauses whose head is *false*. The predicate *false* only occurs in the head of clauses. The formula $\phi_1 \leftarrow \phi_2 \wedge p_1(X_1), \ldots, p_k(X_k)$ is equivalent to the formula $\mathit{false} \leftarrow \neg\phi_1 \wedge \phi_2 \wedge p_1(X_1), \ldots, p_k(X_k)$. The latter might not be a CHC (e.g. if ϕ_1 contains $=$) but can be converted to an equivalent set of CHCs by transforming the formula $\neg\phi_1$ and distributing any disjunctions that arise over the rest of the body. For example, the formula $X = Y \leftarrow p(X, Y)$ is equivalent to the set of CHCs $\{\mathit{false} \leftarrow X > Y, p(X, Y),\ \mathit{false} \leftarrow X < Y, p(X, Y)\}$.

If a set of CHCs encodes the behaviour of a system and the bodies of integrity constraints represent unsafe states, then proving safety consists of showing that the bodies of integrity constraints are unsatisfiable, in which case the integrity constraints are satisfied. In Figure 4.2 the verification problem focuses on proving that the integrity constraint is satisfied. This can only happen if the body of `c3` is unsatisfiable. A program is considered safe if its verification conditions have a model.

4.5.3 *The CHC verification problem.*

To state this more formally, given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. If so we say that P is safe. Obviously any model of P assigns False to the bodies of integrity constraints. We restate this property in terms of the logical consequence relation. Let $P \models F$ mean that F is a logical consequence of P, that is, that every interpretation satisfying P also satisfies F.

Lemma 4.1. *P has a model if and only if $P \not\models \mathit{false}$.*

This lemma holds for arbitrary interpretations (only assuming that the predicate *false* is interpreted as false, uses only the textbook definitions of “interpretation" and “model" and does not depend on the constraint theory. The verification problem can be formulated deductively rather than model-theoretically. We can exploit proof procedures for constraint logic programming [81] to reason about the satisfiability of a set of CHCs.

4.5.3.1 *Proof Techniques for Horn clauses*

The techniques that we use in this chapter are:

- *Proof by over-approximation of the minimal model:* Given a set of CHCs, its minimal model $M[\![P]\!]$ is equivalent to the set of atomic consequences of P [120]. That is, $P \models p(a)$ if and only if $p(a) \in M[\![P]\!]$. Therefore, the CHC verification problem for P is equivalent to checking that $\mathit{false} \notin M[\![P]\!]$. It is sufficient to find a set of constrained facts M' such that $M[\![P]\!] \subseteq M'$, where $\mathit{false} \notin M'$. This technique is called proof by *over-approximation of the minimal model*.
- *Proof by specialisation:* In our context we use specialisation to focus the verification problem on the formula to be proved. More specifically, we specialise a set of CHCs with

respect to a "query" to the atom *false*; thus the specialised CHCs entail *false* if and only if the original clauses entailed *false*. The constraint specialisation procedure described in Section 4.4 is our method of specialisation. So whenever we refer to specialisation we refer to this method unless otherwise stated.

4.5.3.2 *Analysis of the specialised clauses*

Having specialised the clauses with respect to *false*, it may be that the clauses P_{FALSE} do not contain a clause with head *false*. In this case P_{FALSE} is safe, since clearly this is a sufficient condition for $P_{\text{FALSE}} \not\models$ *false*. This is the case for our example program since the body of the clause `c3` is unsatisfiable after constraint strengthening, it is removed from the set of specialised clauses.

If this check fails we still do not know whether P has a model. In this case we can perform the convex polyhedral analysis on the clauses P_{FALSE}. As the experiments later show, safety is often provable by checking the resulting model; if no constrained fact for *false* is present, then $P_{\text{FALSE}} \not\models$ *false*. If safety is not proven, there are two possibilities: the approximate model is not precise enough, but P has a model, or there is a proof of *false*. Refinement techniques could be used to distinguish these, but this is not the topic of this chapter.

In summary, our experimental procedure for evaluating the effectiveness of constraint specialisation contains two steps. Given a set of CHCs P with integrity constraints: (1) Compute a specialisation of P with respect to *false* yielding P_{FALSE}. If P_{FALSE} contains no integrity constraints, then P is safe. (2) If P_{FALSE} does contain integrity constraints, perform a convex polyhedra analysis of P_{FALSE}. If the resulting approximation of the minimal model contains no constrained fact for the predicate *false*, then P_{FALSE} is safe and hence P is safe. If we find a concrete feasible derivation for *false* then we conclude that P is unsafe. Otherwise, P is possibly unsafe. Please refer to [97] for deriving traces using convex polyhedral approximation.

4.6 EXPERIMENTAL EVALUATION

Table 4.1 presents experimental results of applying our constraint specialisation to a number of Horn clause verification benchmarks taken from the repository of Horn clauses [11] and other sources including [62, 84, 68, 12, 37]. The columns CPA, QARMC [64] and Eldarica [80] present the results of verification using convex polyhedra, QARMC and Eldarica respectively, whereas columns CS + CPA, CS + QARMC and CS + Eldarica show the result of running constraint specialisation followed by CPA, QARMC and Eldarica respectively. The symbol "-" is used in the table to indicate that the result is not significant in the given case. The experiments were carried out on an Intel(R) X5355 quad-core (@ 2.66GHz) computer with 6 GB memory running Debian 5. We set 5 minutes of timeout for each experiment. The specialisation procedure is implemented in the tool called RAHFT which is publicly available from `https://github.com/bishoksan/RAHFT/`. The tool offers a simple command line interface and accepts options for constraint specialisation. For this purpose it can be run using the command: `./rahft input -sp output` where the `input` is a set of CHCs and `output` is a file name to store

the specialised CHCs and `-sp` is an option for clause specialisation. The benchmark programs are available from https://github.com/bishoksan/RAHFT/tree/master/benchmarks_scp.

The results show that constraint specialisation is effective in practice. We report that 109 out of 218, that is 50%, of the problems are solved by constraint specialisation alone. When used as a pre-processor for other verification tools, the results show improvements on both the number of instances solved and the solution time. Using our tool, we report approximately 47% increase in the number of instances solved and twice as fast on average. Using QARMC, we report 13% increase in the number of instances solved and 5 times faster on average. Similarly using Eldarica, we report approximately 12% increase in the number of instances solved and almost 4 times faster on average. It is important to note that there is no refinement iteration in CPA as there is in QARMC and Eldarica.

	CPA	CS + CPA	QARMC	CS + QARMC	Eldarica	CS + Eldarica
solved (safe/unsafe)	61 (48/13)	162 (144/18)	178 (141/37)	205 (171/34)	159(135/24)	206 (175/31)
unknown / timeout	144/13	49/7	-/40	-/13	-/59	-/12
total time (secs)	2317	1303	13367	2613	10805	3235
average time (secs)	10.62	5.97	61.31	11.98	50.02	14.97
%solved	27.98	74.31	81.65	94.04	73	95.3

Table 4.1: Experiments on a set of 218 (181 safe and 37 unsafe) CHC verification problems with a timeout of five minutes

The (perhaps surprising) effectiveness of this relatively simple combination of constraint specialisation and convex polyhedral analysis is underlined by noting that it can solve problems for which more complex methods have been proposed. For example, apart from the many examples from the Horn clause verification benchmarks that require refinement using CEGAR-based approaches, the technique solves the “rate-limiter" and “Boustrophedon" examples presented by Monniaux and Gonnord [127] (Section 5) (directly encoded as Horn clauses); their approach, also based on convex polyhedra, uses bounded model checking to achieve a partitioning of the approximation, while other approaches to such problems use trace-partitioning and look-ahead widening.

It is possible to strengthen constraints in the clauses using the model of the original program (denote it by CPA') rather than its query-answer transformed one. The effect of such a specialisation (CS + CPA') on these set of benchmarks is same as applying CPA directly on the original programs, but such a specialisation may be a useful pre-processing for other tools. For example, the following program (a variant of our example program) is not solved by such a combination (CS + CPA') but our current approach does (CS + CPA).

```
false :- A=1, B=0, l(A,B).
l(A,B) :- C=A+B, D=B+1, l(C,D).
l(A,B) :- B>A.
```

We were able to solve almost 100 more problems with our proposed approach. Therefore the role of query-answer transformation is crucial for propagating constraints in verification problems. As mentioned earlier, our specialisation procedure can be iterated which yields

	CPA	CS + CPA
solved (safe/unsafe)	37 (14/23)	61 (26/35)
unknown	95	71
average time (secs.)	0.51	0.50
solved (%)	28	46

Table 4.2: Experimental results on 132 CHC verification problems with a timeout of five minutes

further specialisation of the clauses. By iterating the procedure, we were able to solve 12 more problems. After second iteration, we observed that the same program was produced in most of the cases indicating that the successive iterations do not produce any further specialisation.

4.6.1 *Additional experiments on SV-COMP-15 benchmarks*

We chose a subset of 132 problems, written in C, from SV-COMP 2015[2] [13]. This set contains benchmarks from the categories which were not reported in our experiments before such as *recursive benchmarks* which needs recursive analysis. Additionally it contains some benchmarks from *Loop* category such as *loop-acceleration*, *loop-lit* and *loop-new*. We used SeaHorn [71, 70], a verification framework based on LLVM, for generating Horn clauses from C programs. SeaHorn first compiles C to LLVM intermediate representation (LLVM IR), also known as bitcode using *clang*, a C-family front-end for LLVM[3]. The bitcode is further simplified and optimized reusing the vast amount of work done on LLVM (e.g. function inlining, dead code elimination, CFG simplifications etc.) whose purpose is to make the verification task easier. Gurfinkel et al. [70] have shown that some of the problems are solved by these transformations only. The resulting bitcode is translated to Horn clauses using different semantics for example small step, large block encoding etc. More details can be found in [71, 70]. The benchmark programs are available from `https://github.com/bishoksan/RAHFT/tree/master/benchmarks_scp`. The results are summarised in the Table 4.2. They again show that our method, the constraint specialisation, is in fact effective in practice since we can solve 18% more problems using it as a pre-processor to our convex polyhedra tool.

4.7 CONCLUSION AND FUTURE WORK

We introduced a method for specialising the constraints in constrained Horn clauses with respect to a goal. The method is based a combination of techniques that have already shown their usefulness, especially abstract interpretation and query-answer transformation. The particular combination of techniques we chose was arrived at by experimentation and analysis of

2 http://sv-comp.sosy-lab.org/2015/benchmarks.php
3 http://clang.llvm.org/

the needs of the problem. The approach propagates constraints globally, both forwards and backwards, and produces explicit invariants from the original clauses.

We applied the method to program verification problems encoded as constrained Horn clauses. Experiments showed firstly that constraint specialisation alone is an effective verification tool. Secondly, it can be applied as a pre-processor, improving the effectiveness of other verification tools. It remains to be checked if the solver like VeriMap would benefit from our specialisation.

The effectiveness of the procedure is at first sight somewhat surprising. Its effectiveness comes from the fact that it focuses on full exploitation of the available information, propagating information simultaneously top-down and bottom-up, and the use of powerful analysis techniques based on abstract interpretation capable of discovering useful invariants. Moreover the addition of the widening-upto method with threshold generation plays an important role in the procedure. Care was taken to implement the procedures efficiently.

The technique is independent of the constraint theory underlying the clauses and the abstract domain for analysis, although we only experimented so far with the domain of linear arithmetic constraints, and the domain of convex polyhedra.

FUTURE WORK There is potential for applying this technique in future work whenever explicit constraints need to be extracted from clauses. One such instance is in program debugging since more specific information may make errors in the original program apparent. Another is as a pre-processor in program specialisation where knowledge of the call context of each program point could enable specialisations which are not otherwise obviously available. Finally, termination and resource analysis could benefit from constraint specialisation, since these might enable better ranking functions to be discovered, proving decrease of some expression in each loop.

The query-answer transformation has several variations, which can give differing precision when combined with abstract interpretation. For instance, more refined query predicates of the form $p^q_{i,j}$ could be generated representing calls to the i^{th} atom in the body of clause j [54]. Secondly, the left-to-right computation could be replaced by right-to-left or any other order. The success or failure of a goal is independent of the computation rule; hence we could generate answers using other computation rules, or combining computation rules [55]. While different computation rules do not affect the model of the answer predicates, more effective propagation of constraints during program analysis, and thus greater precision, can sometimes be achieved by varying the computation rule.

ACKNOWLEDGEMENTS

This chapter is an extended version of a paper presented at the ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'15). We thank the anonymous referees of PEPM'15 for useful comments as well as the referees for this journal, who gave many constructive suggestions for improvement. The chapter has been extended in several directions. We have provided algorithms and implementation details, extended the related work and other sections. We have also carried out some further experiments in a set of examples

from software verification benchmarks available in the literature and evaluated our constraint specialisation as a pre-processor to other Horn clause verification tools in the literature. We would like to thank Jorge A. Navas for his help with the tool SeaHorn and for several interesting discussions and Emanuele De Angelis for providing us with benchmark programs. B. Kafle was supported by European Commission Framework 7 project ENTRA (Project 318337). J. Gallagher was supported by Danish Research Council grant FNU 10-084290 “Numeric and Symbolic Abstractions for Software Model Checking".

5

HORN CLAUSE VERIFICATION WITH CONVEX POLYHEDRAL ABSTRACTION AND TREE AUTOMATA-BASED REFINEMENT

With John P. Gallagher

Abstract
In this chapter we apply tree-automata techniques to refinement of abstract interpretation in Horn clause verification. We go beyond previous work on refining trace abstractions; firstly we handle tree automata rather than string automata and thereby can capture traces in any Horn clause derivations rather than just transition systems; secondly, we show how algorithms manipulating tree automata interact with abstract interpretations, establishing progress in refinement and generating refined clauses that eliminate causes of imprecision. We show how to derive a refined set of Horn clauses in which given infeasible traces have been eliminated, using a recent optimised algorithm for tree automata determinisation. We also show how we can introduce disjunctive abstractions selectively by splitting states in the tree automaton. The approach is independent of the abstract domain and constraint theory underlying the Horn clauses. Experiments using linear constraint problems and the abstract domain of convex polyhedra show that the refinement technique is practical and that iteration of abstract interpretation with tree automata-based refinement solves many challenging Horn clause verification problems. We compare the results with other state of the art Horn clause verification tools.

Keywords: Horn clauses, Abstract interpretation, Finite tree automata, Tree automata determinisation.

5.1 INTRODUCTION

The formalism of Constrained Horn clauses (CHCs), as an intermediate language for verification of programs in various languages, has become popular due to its well understood properties and expressiveness; this has led to a range of tools for analysis and verification of CHCs. Given a program and a property ϕ to be verified, a set of CHCs V, such that V is satisfiable if and only if ϕ holds, is called a set of *verification conditions* for ϕ. CHC verification conditions can be obtained from imperative, functional or concurrent languages, among others, by a variety of semantics-based techniques including big- and small-step semantics, Hoare triples, or other intermediate forms such as control-flow graphs [129, 65, 43, 83, 70, 39].

There are several approaches to checking the satisfiability of CHC verification conditions, including abstract interpretation and counterexample-guided abstraction refinement (see Section 5.7). In this chapter we apply tree-automata techniques to refinement of abstract interpretation in Horn clause verification. We go beyond previous work on refining trace abstractions

[74]; firstly, we handle tree automata rather than word automata and thereby can capture traces in any Horn clause derivations rather than just transition systems; secondly, we show how algorithms manipulating tree automata interact with abstract interpretations, establishing progress in refinement and generating refined clauses that eliminate causes of imprecision.

Our approach is similar in spirit to counterexample-guided abstraction refinement (CEGAR) or iterative specialisation approaches, in which a refined set of clauses is generated by eliminating one or more of the infeasible paths from the original set of clauses until the safety or unsafety of the clauses is proven. More specifically, we show how to construct tree automata capturing both the traces (derivations) of a given set of Horn clauses and also one or more infeasible traces discovered after abstract interpretation of the clauses. From these we construct a refined automaton in which the infeasible trace(s) have been eliminated and a new set of clauses is constructed from the refined automaton. This guarantees progress in that the same infeasible trace cannot be generated (in *any* abstract interpretation). In addition, the clauses are restructured during the elimination of the trace, which can lead to more precise abstractions in subsequent iterations. The refinement is manifested in the refined clauses, rather than in an accumulated set of properties as in the CEGAR [23] approach. We rely on the abstract interpretation of the clauses to generate useful properties, rather than hoping to find them during the refinement itself.

We also show how we can introduce disjunctive abstractions selectively by splitting states in the tree automaton. This splitting induces splitting in the predicates of the original set of clauses and its analysis using convex polyhedra leads to disjunctive abstractions. The approach is independent of the abstract domain and constraint theory underlying the Horn clauses. Experiments using linear constraint problems and the abstract domain of convex polyhedra show that the refinement technique is practical and that iteration of abstract interpretation with tree automata-based refinement solves many challenging Horn clause verification problems. We compare the results with other state of the art Horn clause verification tools.

The main contributions are the following; (1) We construct a correspondence between computations using Horn clauses and finite tree automata (FTA) (Section 5.4). (2) We construct a refined set of clauses directly from a tree automaton representation of the clauses and an infeasible trace; the trace is eliminated from the refined clauses (Section 5.4.4). (3) We propose a “splitting" operator on FTAs (Section 5.3) and describe its role in Horn clause verification (Section 5.5.1). (4) We demonstrate the feasibility of our approach in practice applying it to Horn clause verification problems (Section 5.6).

5.2 SUMMARY OF OUR APPROACH

To motivate readers, we present an example set of CHCs P in Figure 5.1 which will be used throughout this chapter. This is an interesting example in which the computations are trees rather than linear sequences.

```
c1. mc91(A,B) :- A > 100, B = A-10.
c2. mc91(A,B) :- A =< 100, C = A+11, mc91(C,D), mc91(D,B).
c3. false :- A =< 100,  B > 91, mc91(A,B).
c4. false :- A =< 100, B =< 90, mc91(A,B).
```

Figure 5.1: Example CHCs. The McCarthy 91-function

After applying abstract interpretation to this set of clauses, we obtain the following set of constained facts (also called approximation). We usually represent a constrained fact derived from abstract interpretation as $p(X) : -[\mathcal{C}(X)]$, where $\mathcal{C}(X)$ is a conjunction of constraints.

```
mc91(A,B) :- [B>90, B>=A-10].
false :- [].
```

Since *false* is in our approximation, our tool generates an abstract derivation for *false* which in our case is the clause `c3` followed by the clause `c1` and is represented by a trace term `c3(c1)`. Since this abstract counterexample is infeasible, our refinement procedure removes this from the set of clauses in Figure 5.1 to produce a new set of clauses as shown in Figure 5.2. Our refinement can be viewed as a program transformation guided by a counterexample. From the set of refined clauses, it can be seen that the counterexample `c3(c1)` is impossible to construct. This refinement split the predicate *mc91* of the original clauses and as a result of this we gain some precision. We again analyse the refined clauses using abstract interpretation until its safety or unsafety is proven. In the sections to follow we describe our abstraction-refinement procedure which led to this result in details.

```
c1: mc91_1(A,B) :-  A>100, B=A-10.
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91_1(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91_1(D,B).
c3: false :- A =< 100, B > 91, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91_1(A,B).
```

Figure 5.2: Refined set of CHCs

The architecture of our abstraction-refinement scheme is shown in Figure 5.3. It is accompanied by our main algorithm 5.1 to give the early picture of our approach.

5.3 FINITE TREE AUTOMATA

Finite tree automata (FTAs) are mathematical machines that define so-called recognisable tree languages, which are possibly infinite sets of terms that have desirable properties such as closure under Boolean set operations and decidability of membership and emptiness.

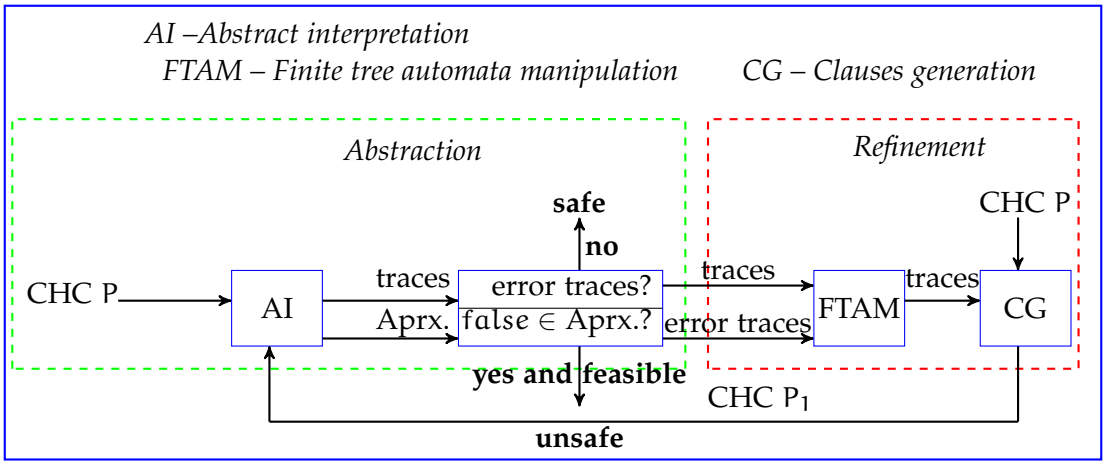

Figure 5.3: *Abstraction-refinement scheme in Horn clause verification. Aprx. is an approximation produced as a result of abstract interpretation.*

Algorithm 5.1: ALGORITHM for abstraction-refinement of Horn clauses

Input: A set of Horn clauses P
Output: *safe* or *unsafe*

1 analyse P using abstract interpretation producing constrained facts M (Algorithm 5.4);
2 if *false* $\notin$ M then **return** *safe* ;
3 if *false* $\in$ M then produce derivation t of *false* using P ;
4 if t is feasible **return** *unsafe* ;
5 $P' \leftarrow$ *refinedCls*(P, t) (Algorithm 5.5) ;
6 P $\leftarrow$ P′ and goto step 1 ;

Definition 5.1 (Finite tree automaton)**.** *An FTA $\mathcal{A}$ is a tuple (Q, Q_f, Σ, Δ), where Q is a finite set of states, $Q_f \subseteq Q$ is a set of final states, Σ is a set of function symbols, and Δ is a set of transitions. We assume that Q and Σ are disjoint.*

Each function symbol $f \in \Sigma$ has an arity $n \geqslant 0$, written as $\mathsf{ar}(f) = n$. The function symbols with arity 0 are called constants. $\mathsf{Term}(\Sigma)$ is the set of ground terms or trees constructed from Σ where $t \in \mathsf{Term}(\Sigma)$ iff $t \in \Sigma$ is a constant or $t = f(t_1, t_2, ..., t_n)$ where $\mathsf{ar}(f) = n$ and $t_1, t_2, ..., t_n \in \mathsf{Term}(\Sigma)$. Similarly $\mathsf{Term}(\Sigma \cup Q)$ is the set of terms/trees constructed from Σ and Q, treating the elements of Q as constants.

Each transition in Δ is of the form $f(q_1, q_2, ..., q_n) \to q$ where $\mathsf{ar}(f) = n$. Given $\delta \in \Delta$ we refer to its left- and right-hand-sides as $\mathsf{lhs}(\delta)$ and $\mathsf{rhs}(\delta)$ respectively. Let $\Rightarrow$ be a one-step rewrite in which $t_1 \Rightarrow t_2$ iff t_2 is the result of replacing one subterm of t_1 equal to $\mathsf{lhs}(\delta)$ by $\mathsf{rhs}(\delta)$, from some $\delta \in \Delta$. The reflexive, transitive closure of $\Rightarrow$ is $\Rightarrow^*$. We say there is a run (resp. successful run) for $t \in \mathsf{Term}(\Sigma)$ if $t \Rightarrow^* q$ where $q \in Q$ (resp. $q \in Q_f$), and we say that t

is *accepted* if t has a successful run. An FTA $\mathcal{A}$ defines a set of terms, that is, a tree language, denoted by $\mathcal{L}(\mathcal{A})$, as the set of all terms accepted by $\mathcal{A}$.

Definition 5.2 (Deterministic FTA (DFTA))**.** *An FTA* (Q, Q_f, Σ, Δ) *is called bottom-up deterministic iff* Δ *has no two transitions with the same left hand side.*

We omit the adjective "bottom-up" in this thesis and just refer to deterministic FTAs. Runs of a DFTA are deterministic in the sense that for every $t \in \mathsf{Term}(\Sigma)$ there is at most one $q \in Q$ such that $t \Rightarrow^* q$.

5.3.1 *Operations on FTAs*

FTAs are closed under Boolean set operations, but for our purposes we mention only union and difference of language of automata, where in addition we assume that the signature Σ is fixed and that the states of FTAs are disjoint from each other when applying operations (the states can be renamed apart).

Definition 5.3 (Union of FTAs)**.** *Let* $\mathcal{A}^1, \mathcal{A}^2$ *be FTAs* $(Q^1, Q^1_f, \Sigma, \Delta^1)$ *and* $(Q^2, Q^2_f, \Sigma, \Delta^2)$ *respectively. Then* $\mathcal{A}^1 \cup \mathcal{A}^2 = (Q^1 \cup Q^2, Q^1_f \cup Q^2_f, \Sigma, \Delta^1 \cup \Delta^2)$*, and we have* $\mathcal{L}(\mathcal{A}^1 \cup \mathcal{A}^2) = \mathcal{L}(\mathcal{A}^1) \cup \mathcal{L}(\mathcal{A}^2)$.

Determinisation plays a key role in the theory of FTAs. As far as expressiveness is concerned, we can limit our attention to DFTAs since for every FTA $\mathcal{A}$ there exists a DFTA $\mathcal{A}^d$ such that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}^d)$ [28]. The standard construction builds a DFTA $\mathcal{A}^d$ whose states are elements of the powerset of the states of $\mathcal{A}$. The textbook procedure for constructing $\mathcal{A}^d$ from $\mathcal{A}$ [28] is not viewed as a practical procedure for manipulating tree automata, even fairly small ones. In a recent work Gallagher *et al.* [60] developed an optimised algorithm for determinisation, whose worst-case complexity remains unchanged, but which performs dramatically better than existing algorithms in practice. A critical aspect of the algorithm is that the transitions of the determinised automaton are generated in a potentially very compact form called *product form*, which can often be used directly when manipulating the determinised automaton.

Definition 5.4 (Product Transition)**.** *A* product transition *is of the form* $f(Q_1, \ldots, Q_n) \to q$ *where* Q_i *are sets of states and* q *is a state. The product transition represents a set of transitions* $\{f(q_1, \ldots, q_n) \to q \mid q_i \in Q_i, i = 1..n\}$*. Thus* $\Pi_{i=1}^{n} |Q_i|$ *transitions are represented by a single product transition.*

Alternatively, we can regard a product transition as introducing ϵ-transitions. An ϵ-transition has the form $q_1 \to q_2$ where q_1, q_2 are states. ϵ-transitions can be eliminated, if desired. Given a product transition $f(Q_1, \ldots, Q_n) \to q$, introduce n new non-final states $s_1, \ldots, s_n$ corresponding to $Q_1, \ldots, Q_n$ respectively and replace the product transition by the set of transitions $\{f(s_1, \ldots, s_n) \to q\} \cup \{q' \to s_i \mid q' \in Q_i, 1 = 1..n\}$. It can be shown that this transformation preserves the language of the FTA.

Given FTAs $\mathcal{A}^1$ and $\mathcal{A}^2$ there exists an FTA $\mathcal{A}^1 \setminus \mathcal{A}^2$ such that $\mathcal{L}(\mathcal{A}^1 \setminus \mathcal{A}^2) = \mathcal{L}(\mathcal{A}_1) \setminus \mathcal{L}(\mathcal{A}^2)$. To construct the difference FTA we use union and determinisation and exploit the following property of determinised states [60].

Property 5.1. *Let $\mathcal{A}^d$ be the DFTA constructed from $\mathcal{A}$. Let Q be the states of $\mathcal{A}$. Then there is a run $t \Rightarrow^* q$ in $\mathcal{A}$ if and only if there exists Q' and a run $t \Rightarrow^* Q'$ in $\mathcal{A}^d$ where $Q' \in 2^Q$, such that $q \in Q'$.*

Furthermore recall that a term is accepted by at most one state in a DFTA. This gives rise to the following construction of the difference FTA $\mathcal{A}^1 \setminus \mathcal{A}^2$. We first form the DFTA for the union of the two FTAs and then remove those of its final states containing the final states of $\mathcal{A}^2$. In this way we remove the terms, and only the terms (by Property 5.1), accepted by $\mathcal{A}^2$. The availability of a practical algorithm for determinisation is what makes this construction of the difference FTA feasible.

Definition 5.5 (Construction of difference of FTAs)**.** *Let $\mathcal{A}^1, \mathcal{A}^2$ be FTAs $(Q^1, Q^1_f, \Sigma, \Delta^1)$ and $(Q^2, Q^2_f, \Sigma, \Delta^2)$ respectively. Let $(\mathcal{Q}', \mathcal{Q}'_f, \Sigma, \Delta')$ be the determinisation of $\mathcal{A}^1 \cup \mathcal{A}^2$. Let $\mathcal{Q}^2 = \{Q' \in \mathcal{Q}' \mid Q' \cap Q^2_f \neq \emptyset\}$. Then $\mathcal{A}^1 \setminus \mathcal{A}^2 = (\mathcal{Q}', \mathcal{Q}'_f \setminus \mathcal{Q}^2, \Sigma, \Delta')$.*

Next we introduce a new operation over FTA called *state splitting*, which consists of splitting a state q into a number of states, based on a partition of the set of transitions whose rhs is q. We define this splitting as follows:

Definition 5.6 (Splitting a state in an FTA)**.** *Let $\mathcal{A} = (Q, Q_f, \Sigma, \Delta)$ be an FTA. Let $q \in Q$ and $\Delta_q = \{t \in \Delta \mid \mathsf{rhs}(t) = q\}$. Let $\Phi = \{\Delta^1_q, \ldots, \Delta^k_q\}$ $(k > 1)$ be some partition of Δ_q. Introduce k new states $q_1, \ldots, q_k$. Then the FTA $\mathsf{split}_\Phi(\mathcal{A})$ is $(Q^s, Q^s_f, \Sigma, \Delta^s)$ where:*

- $Q^s = (Q \setminus \{q\}) \cup \{q_1, \ldots, q_k\}$;
- $Q^s_f = (Q_f \setminus \{q\}) \cup \{q_1, \ldots, q_k\}$ *if* $q \in Q_f$, *otherwise* $Q^s_f = Q_f$;
- $\Delta^s = \mathsf{unfold}_q(\Delta \setminus \Delta_q \cup \{\mathsf{lhs}(t) \to q_i \mid t \in \Delta^i_q, i = 1..k\})$, *where* $\mathsf{unfold}_q(\Delta')$ *is the result of repeatedly replacing a transition* $f(\ldots, q, \ldots) \to s \in \Delta'$ *by the set of* k *transitions* $\{f(\ldots, q_1, \ldots) \to s, \ldots, f(\ldots, q_k, \ldots) \to s\}$ *until no more such replacements can be made.*

Proposition 5.1. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathsf{split}_\Phi(\mathcal{A}))$.

Proof. Let $\mathcal{A} = \langle Q, Q_f, \Sigma, \Delta\rangle$ and $\mathsf{split}_\Phi(\mathcal{A}) = \langle Q^s, Q^s_f, \Sigma, \Delta^s\rangle$. Let $\mathsf{split}(q)$ mean that the state q is split and let $q_1, \ldots, q_k$ be the new states introduced during splitting. We write $\Rightarrow^*$ for derivations in $\mathcal{A}$ and $\Rightarrow^*_s$ for derivations in $\mathsf{split}_\Phi(\mathcal{A})$. We first prove by induction on the depth of terms that for all terms t and states $q \in Q$,

$$(\mathsf{split}(q) \to (t \Rightarrow^* q \equiv \exists i.(t \Rightarrow^*_s q_i))) \wedge (\neg\mathsf{split}(q) \to (t \Rightarrow^* q \equiv t \Rightarrow^*_s q)). \tag{1}$$

Base case. Let a be a term of depth 1.

$$\begin{aligned}
&\mathsf{split}(q) \to \\
&a \Rightarrow^* q \quad &\equiv\ & a \to q \in \Delta \wedge \exists\, i.(q \in \Delta^i_q \wedge a \to q_i \in \Delta^s) \\
& &&\text{following Definition 5.6} \\
& &\equiv\ & \exists i.(a \Rightarrow^*_s q_i) \\
&\neg\mathsf{split}(q) \to \\
&a \Rightarrow^* q \quad &\equiv\ & a \to q \in \Delta \wedge a \to q \in \Delta^s \\
& &\equiv\ & a \Rightarrow^*_s q)
\end{aligned}$$

Inductive case. Let $f(t_1,\ldots,t_n) \Rightarrow q$ where $f(t_1,\ldots,t_n)$ is a term of depth $k+1$ and assume that the property holds for all terms with depth at most k.

$\mathsf{split}(q) \rightarrow$

$$
\begin{aligned}
f(t_1,\ldots,t_n) \Rightarrow^* q \quad &\equiv \quad \exists\, r_1,\ldots,r_n\,.(f(r_1,\ldots,r_n) \rightarrow q \in \Delta \wedge \\
&\quad\; t_1 \Rightarrow^* r_1 \;\wedge\cdots\wedge\, t_n \Rightarrow^* r_n) \wedge \\
&\quad\; (\mathsf{split}(r_1) \vee \neg\mathsf{split}(r_1)) \wedge\cdots\wedge (\mathsf{split}(r_n) \vee \neg\mathsf{split}(r_n)) \\
&\equiv \quad \exists\, r_1,\ldots,r_n\,.[f(r_1,\ldots,r_n) \rightarrow q \in \Delta \wedge \\
&\quad\; (\mathsf{split}(r_1) \wedge \exists i_1 (t_1 \Rightarrow_s^* r_{1,i_1})) \vee (\neg\mathsf{split}(r_1) \wedge t_1 \Rightarrow_s^* r_1) \wedge \\
&\quad\; \ldots \\
&\quad\; (\mathsf{split}(r_n) \wedge \exists i_n (t_n \Rightarrow_s^* r_{n,i_n})) \vee (\neg\mathsf{split}(r_n) \wedge t_n \Rightarrow_s^* r_n))] \\
&\quad\; \text{by inductive hypothesis after rearranging formula, since } t_1,\ldots,t_n \\
&\quad\; \text{have depth at most } k \\
&\equiv \quad \exists\, r_1,\ldots,r_n\; \exists i_1,\ldots,i_n.[f(r_1,\ldots,r_n) \rightarrow q \in \Delta \wedge \\
&\quad\; ((t_1 \Rightarrow_s^* r_{1,i_1} \vee t_1 \Rightarrow_s^* r_1) \;\wedge\cdots\wedge (t_n \Rightarrow_s^* r_{n,i_n} \vee t_n \Rightarrow_s^* r_n))] \\
&\quad\; \text{after rearranging formula, moving quantifiers outwards} \\
&\quad\; \text{and eliminating } \mathsf{split}(r_i) \vee \neg\mathsf{split}(r_i), 1 \leqslant i \leqslant n \\
&\equiv \quad \exists\, i, r_1,\ldots,r_n\; \exists i_1,\ldots,i_n.[f(r_{1,i_1},\ldots,r_{n,i_n}) \rightarrow q_i \in \Delta^s \wedge \\
&\quad\; ((t_1 \Rightarrow_s^* r_{1,i_1} \vee (r_{1,i_1} = r_1 \wedge t_1 \Rightarrow_s^* r_1)) \;\wedge\cdots\wedge (t_n \Rightarrow_s^* r_{n,i_n} \vee \\
&\quad\; (r_{n,i_n} = r_n \wedge t_n \Rightarrow_s^* r_n)))] \\
&\quad\; \text{applying Definition 5.6 to introduce } f(r_{1,i_1},\ldots,r_{n,i_n}) \rightarrow q_i \\
&\quad\; \text{since } f(r_{1,i_1},\ldots,r_{n,i_n}) \rightarrow q_i \text{ is included after applying unfold to} \\
&\quad\; f(r_1,\ldots,r_n) \rightarrow q_i \\
&\equiv \quad \exists\, i. f(t_1,\ldots,t_n) \Rightarrow_s^* q_i
\end{aligned}
$$

$\neg\mathsf{split}(q) \rightarrow$

$$
\begin{aligned}
f(t_1,\ldots,t_n) \Rightarrow^* q \quad &\equiv \quad [\text{Similar to previous case but where } f(r_{1,i_1},\ldots,r_{n,i_n}) \rightarrow q \\
&\quad\; \text{is in the unfolding of } f(r_1,\ldots,r_n) \rightarrow q \text{ in Definition 5.6}] \\
&\equiv \quad f(t_1,\ldots,t_n) \Rightarrow_s^* q
\end{aligned}
$$

Finally, if $q \in Q_f$ and $\mathsf{split}(q)$ then $q_i \in Q_f^s$ where q_i is a new state introduced during the splitting. It follows from this and property (1) that for all t, $\exists q \in Q_f\,.\, t \Rightarrow^* q \equiv \exists q' \in Q_f^s\,.\, t \Rightarrow_s^* q'$. Thus for all t, $t \in \mathcal{L}(\mathcal{A})$ iff $t \in \mathcal{L}(\mathsf{split}_\Phi(\mathcal{A}))$.

□

5.4 HORN CLAUSES AND THEIR TRACE AUTOMATA

A constrained Horn clause (CHC) is a first order predicate logic formula of the form $\forall(\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k) \rightarrow p(X))$ ($k \geqslant 0$), where ϕ is a conjunction of constraints with respect

to some constraint theory, X_i, X are (possibly empty) vectors of distinct variables, $p_1, \ldots, p_k, p$ are predicate symbols, $p(X)$ is the head of the clause and $\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k)$ is the body.

There is a distinguished predicate symbol *false* which is interpreted as FALSE. In practice the predicate *false* only occurs in the head of clauses; we call clauses whose head is *false* *integrity constraints*, following the terminology of deductive databases. They are also sometimes referred to as negative clauses. We follow the syntactic conventions of constraint logic programs and write a clause as $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$.

5.4.1 *Interpretations and models*

An interpretation of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \phi$ where A is an atomic formula $p(Z_1, \ldots, Z_n)$ where $Z_1, \ldots, Z_n$ are distinct variables and ϕ is a constraint over $Z_1, \ldots, Z_n$. This set may be infinite. The constrained fact $A \leftarrow \phi$ is shorthand for the set of variable-free facts $A\theta$ such that $\phi\theta$ holds in the constraint theory, and an interpretation M denotes the set of all facts denoted by its elements; M assigns true to exactly those facts. $M_1 \subseteq M_2$ if the set of denoted facts of M_1 is contained in the set of denoted facts of M_2.

MINIMAL MODELS A model of a set of CHCs is an interpretation that satisfies (whenever the body of a clause holds under the given interpretation then so does the head) each clause. There exists a minimal model with respect to the subset ordering, denoted $M[\![P]\!]$ where P is a satisfiable set of CHCs. $M[\![P]\!]$ can be computed as the least fixed point (lfp) of an immediate consequences operator (called S^D_P in [81, Section 4]), which is an extension of the standard T_P operator from logic programming, extended to handle the constraint theory D. Furthermore $\mathrm{lfp}(S^D_P)$ can be computed as the limit of the ascending sequence of interpretations $\emptyset, S^D_P(\emptyset), S^D_P(S^D_P(\emptyset)), \ldots$. This sequence provides a basis for abstract interpretation of CHC clauses. The minimal model of P is equivalent to the set of atomic logic consequences of P.

5.4.2 *The constrained Horn clause verification problem.*

Given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. Obviously any model of P assigns false to the bodies of integrity constraints. We restate this property in terms of the derivability ($\vdash$) of the predicate *false*. Let $P \models F$ mean that F is a logical consequence of P, that is, that every interpretation satisfying P also satisfies F.

Lemma 5.1. *P has a model if and only if $P \not\models$ false.*

This lemma holds for arbitrary interpretations (only assuming that the predicate *false* is interpreted as FALSE), uses only the textbook definitions of “interpretation" and “model" and does not depend on the constraint theory. We have yet another equivalent formulation of the CHC verification problem.

Lemma 5.2. P *has a model if and only if* $\mathsf{P} \not\vdash$ *false.*

Proof. Follows from the equivalence of the minimal model of P with the set of atomic logical consequences of P [81]. See also Proposition 5.2 later. □

An assertion ϕ is an invariant (over-approximation) for a predicate q in P, if $\mathsf{P} \models \forall(q \to \phi)$. If a set of Horn clauses P have a model then we say that P is *safe*, otherwise we say that P is *unsafe*.

5.4.3 *Trace automata for CHCs*

Before constructing the trace automaton we introduce identifiers for each clause. An identifier is a function symbol whose arity is the same as the number of atoms in the clause body. For instance a clause $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$ is assigned a function symbol with arity k. More than one clause can be assigned the same function symbol, but all the clauses with the same identifier have the same structure, including their constraints; that is, they differ only in one or more predicate names. Given a set of CHCs and a set Σ of ranked function symbols, let $\mathsf{id}_\mathsf{P} : \mathsf{P} \to \Sigma$ be the assignment of function symbols to clauses.

Definition 5.7 (Trace FTA for a set of CHCs)**.** *Let* P *be a set of CHCs. Define the trace FTA for* P *as* $\mathcal{A}_\mathsf{P} = (Q, Q_f, \Sigma, \Delta)$ *where*

- Q *is the set of predicate symbols of* P*;*
- $Q_f = Q$*;*
- Σ *is a set of function symbols;*
- $\Delta = \{c(p_1, \ldots, p_k) \to p \mid \text{where } c \in \Sigma,\ c = \mathsf{id}_\mathsf{P}(cl), \text{where } cl = p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)\}$.

The elements of $\mathcal{L}(\mathcal{A}_\mathsf{P})$ *are called trace terms for* P*. In Section 5.5 we will see that several clauses differing only in their predicate names are assigned the same function symbol.*

Example 5.1. *Let* P *be the set of CHCs in Figure 5.1. Let* id_P *map the clauses to* $c_1, \ldots, c_4$ *respectively. Then* $\mathcal{A}_\mathsf{P} = (Q, Q_f, \Sigma, \Delta)$ *where:*

$$
\begin{array}{lcllcl}
Q & = & \{\texttt{mc91}, \texttt{false}\} & \Delta & = & \{c_1 \to \texttt{mc91}, \\
Q_f & = & \{\texttt{mc91}, \texttt{false}\} & & & c_2(\texttt{mc91}, \texttt{mc91}) \to \texttt{mc91}, \\
\Sigma & = & \{c_1, c_2, c_3, c_4\} & & & c_3(\texttt{mc91}) \to \texttt{false},\ c_4(\texttt{mc91}) \to \texttt{false}\}
\end{array}
$$

For each trace term there exists a corresponding derivation tree called an AND-tree, which is unique up to variable renaming. The concept of an AND-tree is derived from [142] and [56].

Definition 5.8 (AND-tree for a trace term)**.** *Let* P *be a set of CHCs and let* $t \in \mathcal{L}(\mathcal{A}_\mathsf{P})$*. Denote by* $\mathsf{AND}(t)$ *the following labelled tree, where each node of* $\mathsf{AND}(t)$ *is labelled by a clause and an atomic formula.*

1. *For each subterm* $c_j(t_1, \ldots, t_k)$ *of* t *there is a corresponding node in* $\mathsf{AND}(t)$ *labelled by an atom* $p(X)$ *and an identifier* c_j *of the clause* $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$ *which is a renamed version of some clause* c *in* P*, such that* $c_j = \mathsf{id}_P(c)$*; the node's children (if* $k > 0$*) are the nodes corresponding to* $t_1, \ldots, t_k$ *and are labelled by* $p_1(X_1), \ldots, p_k(X_k)$*.*
2. *The variables in the labels are chosen such that if a node* n *is labelled by a clause, the local variables in the clause body do not occur outside the subtree rooted at* n*.*

Definition 5.9 (Trace constraints)**.** *Let* P *be a set of CHCs. The set of constraints of a trace* $t \in \mathcal{L}(\mathcal{A}_P)$*, represented as* $\mathsf{constr}(t)$ *is the set of all constraints in the clause labels of* $\mathsf{AND}(t)$*.*

Definition 5.10 (Feasible trace)**.** *We say that a trace term* t *is feasible if* $\mathsf{constr}(t)$ *is satisfiable.*

Definition 5.11 (FTA for a trace term)**.** *Let* P *be a set of CHCs and* $t \in \mathcal{L}(\mathcal{A}_P)$*. The FTA* A_t *(whose construction is trivial) such that* $\mathcal{L}(\mathcal{A}_t) = \{t\}$ *is called* the FTA for t*. The states of* $\mathcal{A}_t$ *are chosen to be disjoint from those of* $\mathcal{A}_P$*.*

Example 5.2 (Trace FTA)**.** *Consider the FTA in Example 5.1. Let* $t = c_3(c_2(c_1, c_1))$*. Each* e_i *represents a label in the trace. Then* $\mathcal{A}_t = (Q, Q_f, \Sigma, \Delta)$ *is defined as:*

$$\begin{aligned} Q &= \{e_1, e_2, e_3, e_4\} \\ Q_f &= \{e_1\} \\ \Sigma &= \{c_1, c_2, c_3, c_4\} \\ \Delta &= \{c_1 \to e_3,\ c_1 \to e_4,\ c_2(e_3, e_4) \to e_2, \\ &\qquad c_3(e_2) \to e_1\} \end{aligned}$$

and Σ *is the same as in* $\mathcal{A}_P$*. The trace* t *is not feasible since*

$$\mathsf{constr}(t) = \{A \leqslant 100, B > 91, A \leqslant 100, C = A + 11, C > 100, D = C - 10, D > 100, B = D - 10\}$$

and this is not satisfiable.

Example 5.3 (Trace FTA of a linear trace)**.** *Consider the FTA in Example 5.1 and a linear trace* $t = c_3(c_1)$*. Let* e_1 *and* e *represents the labels in the trace. Then* $\mathcal{A}_t = (Q, Q_f, \Sigma, \Delta)$ *is defined as:*

$$\begin{aligned} Q &= \{e, e_1\} \\ Q_f &= \{e\} \\ \Sigma &= \{c_1, c_2, c_3, c_4\} \\ \Delta &= \{c_1 \to e,\ c_3(e_1) \to e\} \end{aligned}$$

and Σ *is the same as in* $\mathcal{A}_P$*. The trace* t *is not feasible.*

Definition 5.12 (Constrained trace atom)**.** *Let* P *be a set of CHCs and* $t \in \mathcal{L}(\mathcal{A}_P)$*. Let* $p(X)$ *be the atom labeling the root of* $\mathsf{AND}(t)$*. Then the constrained trace atom of* t *is* $\forall X.(\exists \bar{Z}.\mathsf{constr}(t) \to p(X))$*, where* $\bar{Z} = \mathsf{vars}(\mathsf{constr}(t)) \setminus X$*.*

We now restate standard soundness and completeness results from constraint logic programming [81] in terms of the concepts defined above. We assume that the underlying constraint theory $\mathcal{T}$ has a complete satisfiability procedure. Note that the domain of linear arithmetic constraints, which is used in our experiments, satisfies these conditions.

Proposition 5.2. *Let P be a set of CHCs, whose underlying constraint theory $\mathcal{T}$ has a complete satisfiability procedure. Let $\mathcal{A}_P$ be the trace FTA for P. Then*

1. *for all $t \in \mathcal{L}(\mathcal{A}_P)$, $P \cup \mathcal{T} \models A$, where A is the constrained trace atom of t;*
2. *$p(c)$ is a variable-free atomic formula such that $P \cup \mathcal{T} \models p(c)$*
   a) *iff there exists a feasible trace $t \in \mathcal{L}(\mathcal{A}_P)$ whose constrained trace atom is of the form $\forall X.\phi \to p(X)$ where the constraint $\phi[X/c]$ is true.*
   b) *iff $p(c)$ is in $M[\![P]\!]$, the minimal model of P.*

Proof. The proof depends on a close correspondence between AND-trees (Definition 5.8) and derivations defined as sequences in [81]; we do not elaborate the correspondence in detail but just note that for each AND-tree with constrained trace atom $\forall X.(\exists \bar{Z}.\mathsf{constr}(t) \to p(X))$ there exists one or more derivations for $p(X)$ with answer constraint $\exists \bar{Z}.\mathsf{constr}(t)$. Conversely for each derivation for $p(X)$ with answer constraint ϕ there exists a unique AND-tree whose root is labelled with $p(X)$ and whose constrained trace atom is $\forall X.\phi \to p(X)$.

(1). Let $\mathsf{AND}(t)$ be the AND-tree for t (Definition 5.8), let $p(X)$ be the atom labeling the root and let $\forall X.\phi \to p(X)$ be the constrained trace atom for t.

$\Rightarrow$ exists a derivation (as defined in [81]) for $p(X)$ having answer constraint $\phi \to p(X)$;

$\Rightarrow$ $P \cup \mathcal{T} \models \phi \to p(X)$ by [81] (Theorem 6.1, Part 2).

2(a). $P \cup \mathcal{T} \models p(c)$ is equivalent to $P \cup \mathcal{T} \models X = c \to p(X)$.

$\equiv$ there is a derivation for $p(X)$ with answer constraint ϕ,
where $\mathcal{T} \models X = c \to \phi$ (by [81] (Theorem 6.1, Part 4));

$\equiv$ there is a trace term t and AND-tree $\mathsf{AND}(t)$
with root labelled by $p(X)$ and constrained trace atom $\forall X.\phi \to p(X)$, where $\mathcal{T} \models \phi[X/c]$.

2(b). Follows directly from [81] (Theorem 6.1, Part 1,2)

□

For part 2(a) of the proof, note that the constrained trace atom in the AND-tree can be more general than the atom $p(c)$. For example, say that $p(1)$ is a consequence of the set of CHCs; then the constrained trace atom could be $\forall X.X \geqslant 0 \to p(X)$.

Proposition 5.2 establishes the correspondence between the semantics of CHCs and the feasible traces of the trace FTA for the CHCs. Essentially, the set of feasible traces of the FTA is a representation of the minimal model of the clauses.

If we transform $\mathcal{A}_P$ to another FTA while preserving the set of traces, we also preserve the feasible traces. More generally, we can transform $\mathcal{A}_P$ to another FTA $\mathcal{A}'$ so long as $\mathcal{L}(\mathcal{A}') \subseteq \mathcal{L}(\mathcal{A}_P)$ and the elements of $\mathcal{L}(\mathcal{A}_P) \setminus \mathcal{L}(\mathcal{A}')$ are all infeasible. In this case the feasible traces of $\mathcal{L}(\mathcal{A}')$ are still a representation of the minimal model of P. We will exploit this in our refinement procedure (see Section 5.5).

5.4.4 *Generation of CHCs from a trace FTA*

Now we describe a procedure (Algorithm 5.2) for generating a set of clauses P' from an FTA $\mathcal{A} = (Q, Q_f, \Sigma, \Delta)$ and a set of clauses P. We assume that Σ is the same as that of $\mathcal{A}_P$; so Σ is the range of the function id_P mapping clauses of P to function symbols. The transitions Δ are not in product form; a modification of the algorithm and its correctness proposition is possible for product form (which is in fact an enabling factor which makes possible the determinisation of FTAs in practice) but we omit that here for the simplicity of presentation. We first introduce an injective function for renaming the states of $\mathcal{A}$ since we need predicate names for the generated clauses.

$$\rho_1 : Q \to \mathsf{Predicates}$$

The function ρ_1 maps each FTA state to a distinct predicate name. The algorithm simply generates a clause for each transition, applying the renaming function from states to predicates, and introducing variables arguments according to the pattern obtained from any clause with the corresponding identifier (all clauses with the same identifier having the same variable pattern).

Algorithm 5.2: Generating a set of clauses represented by an FTA

Input: An FTA $\mathcal{A} = (Q, Q_f, \Sigma, \Delta)$, set of Horn clauses P, injective functions $\rho_1 : Q \to \mathsf{Predicates}$, $\mathrm{id}_P : P \to \Sigma$

Output: A set of Horn clauses P' represented by $\mathcal{A}$ and function $\mathrm{id}_{P'} : P' \to \Sigma$

1 $P' \leftarrow \emptyset$;
2 **for** *each* $c_i(q_1, \ldots, q_n) \to q$ *(where* $n \geqslant 0$*)* $\in \Delta$ **do**
3 let $c = p(X) \leftarrow \phi, p_1(X_1), \ldots, p_n(X_n)$ be the clause in P where $\mathrm{id}_P(c) = c_i$;
4 $c_{new} = \rho_1(q)(X) \leftarrow \phi, \rho_1(q_1)(X_1), \ldots, \rho_1(q_n)(X_n)$;
5 $\mathrm{id}_{P'}(c_{new}) = c_i$;
6 $P' \leftarrow P' \cup \{c_{new}\}$;
7 **return** P';

Apart from generating a set of clauses P', Algorithm 5.2 also generates the clause identification mapping $\mathrm{id}_{P'}$, preserving the function symbols from the FTA. In this way the set of traces is preserved from P to P'. The correctness of Algorithm 5.2 is expressed by the following proposition.

Proposition 5.3. *Let P be a set of CHCs and let $\mathcal{A}$ be an FTA whose signature is the same as that of $\mathcal{A}_P$. Let P' be the set of clauses generated from $\mathcal{A}$ and P by Algorithm 5.2. Then $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_{P'})$.*

Furthermore if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_P)$ and $\mathcal{L}(\mathcal{A})$ includes all the feasible traces of $\mathcal{L}(\mathcal{A}_P)$ then the minimal model of P' is the same as the minimal model of P, modulo predicate renaming.

Proof. We first prove that $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_{P'})$, that is, $\forall t \,.\, t \in \mathcal{L}(\mathcal{A}) \equiv t \in \mathcal{L}(\mathcal{A}_{P'})$. The proof is by induction on the depth of t. Let $\mathcal{A} = \langle Q, Q_f, \Sigma, \Delta\rangle$ and $\mathcal{A}_{P'} = \langle Q', Q'_f, \Sigma, \Delta'\rangle$ and we assume that $Q = Q_f$ and $Q' = Q'_f$.

- Base case.

$$\begin{aligned}
t \text{ has depth } 0 \text{ and } t \in \mathcal{L}(\mathcal{A}) &\equiv \exists\, t \to q \in \Delta \\
&\equiv \exists\, c = p(X) \leftarrow \phi \in P \text{ where } id_P(c) = t \\
&\equiv \exists\, c_{new} = \rho_1(q)(X) \leftarrow \phi \in P' \text{ and } id_{P'}(c_{new}) = t \\
&\equiv \exists\, t \to \rho_1(q) \in \Delta' \\
&\equiv t \in \mathcal{L}(\mathcal{A}_{P'})
\end{aligned}$$

- Inductive case. Assume that for all terms t of depth at most k, $t \Rightarrow^* q$ in $\mathcal{A}$ iff $t \Rightarrow^* \rho_1(q)$ in $\mathcal{A}_{P'}$. Let $t = c_t(t_1, \ldots, t_n)$ have depth $k+1$.

$$\begin{aligned}
c_t(t_1, \ldots, t_n) \in \mathcal{L}(\mathcal{A}) &\equiv \exists\, c_t(q_1, \ldots, q_n) \to q \in \Delta \wedge t_i \Rightarrow^* q_i, 1 \leqslant i \leqslant n \\
&\equiv \exists\, c_t(q_1, \ldots, q_n) \to q \in \Delta \wedge t_i \Rightarrow^* \rho_1(q_i), 1 \leqslant i \leqslant n \\
&\quad \text{by ind. hyp. since depth of } t_1, \ldots t_n \text{ is at most } k \\
&\equiv \exists\, c = p(X) \leftarrow \phi, p_1(X_1), \ldots, p_n(X_n) \in P \text{ where } id_P(c) = c_t \\
&\quad \wedge t_i \Rightarrow^* \rho_1(q_i), 1 \leqslant i \leqslant n \\
&\equiv \exists\, c_{new} = \rho_1(q)(X) \leftarrow \phi, \rho_1(q_1)(X_1), \ldots, \rho_1(q_n)(X_n) \in P' \\
&\quad \text{and } id_{P'}(c_{new}) = c_t \\
&\quad \wedge t_i \Rightarrow^* \rho_1(q_i), 1 \leqslant i \leqslant n \\
&\equiv \exists\, c_t(\rho_1(q_1), \ldots, \rho_1(q_n)) \to \rho(q) \in \Delta' \\
&\quad \wedge t_i \Rightarrow^* \rho_1(q_i), 1 \leqslant i \leqslant n \\
&\equiv c_t(t_1, \ldots, t_n) \in \mathcal{L}(\mathcal{A}_{P'})
\end{aligned}$$

This completes the proof that $\mathcal{L}(\mathcal{A}_{P'}) = \mathcal{L}(\mathcal{A})$. Now assume that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{A}_P)$ and includes all the feasible traces of $\mathcal{L}(\mathcal{A}_P)$; that is, for all feasible traces t, $t \in \mathcal{L}(\mathcal{A}_P)$ iff $t \in \mathcal{L}(\mathcal{A})$. Then by Proposition 5.2 $M[\![P]\!] = M[\![P']\!]$ since $\mathcal{A}$ is the trace FTA for P'. □

Example 5.4 (Generation of clauses from an FTA)**.** *Consider the following transitions, relating to the signature for the program in Figure 5.1 (this FTA can be the result of applying automata based transformations to the FTA corresponding to the program in Figure 5.1). The set of states is* `{[false],[mc91],[e,false],[mc91,e1]}`*. (These are elements of the powerset of the set of states* `{false,mc91,e,e1}` *obtained from the union of FTA in Example 5.1 and FTA in Example 5.3, which were generated by the determinisation algorithm).*

```
c1 -> [mc91, e1].
c2([mc91, e1],[mc91, e1]) -> [mc91].
c2([mc91],[mc91]) -> [mc91].
c2([mc91, e1],[mc91]) -> [mc91].
c2([mc91],[mc91, e1]) -> [mc91].
c3([mc91]) -> [false].
c4([mc91, e1]) -> [false].
c4([mc91]) -> [false].
c3([mc91, e1]) -> [e, false].
```

The clauses generated by Algorithm 5.2 are the following, with the renaming function $\rho_1 = \{[\texttt{false}] \mapsto \texttt{false}, [\texttt{mc91}] \mapsto \texttt{mc91}, [\texttt{e}, \texttt{false}] \mapsto \texttt{false\_1}, [\texttt{mc91}, \texttt{e1}] \mapsto \texttt{mc91\_1}\}$*. Below we also show the clause identifiers (the* id *function for the generated clauses) showing that several clauses can have the same identifier, thus preserving traces.*

```
c1: mc91_1(A,B) :-  A>100, B=A-10.
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91_1(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91_1(C,D),  mc91(D,B).
c2: mc91(A,B) :-  A=<100,  C=A+11, mc91(C,D),  mc91_1(D,B).
c3: false :- A =< 100, B > 91, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91(A,B).
c4: false :- A =< 100, B =< 90, mc91_1(A,B).
c3: false_1 :- A =< 100, B > 91, mc91_1(A,B).
```

5.4.5 *Abstract Interpretation of Constrained Horn Clauses*

Abstract interpretation [32] is a technique which derives sound over-approximations by computing abstract fixed points. Convex polyhedron analysis (CPA) [31] is a program analysis technique based on abstract interpretation [32]. When applied to a set of CHCs P it constructs an over-approximation M' of the minimal model of P, where M' contains at most one constrained fact $p(X) \leftarrow \phi$ for each predicate p. The constraint ϕ is a conjunction of linear inequalities, representing a convex polyhedron. The first application of convex polyhedron analysis to CHCs was by Benoy and King [10].

We summarise briefly the elements of convex polyhedron analysis for CHC; further details (with application to CHC) can be found in [31, 10]. The abstract interpretation consists of the computation of an increasing sequence of elements of the abstract domain of tuples of convex polyhedra (one for each predicate) $\mathcal{D}^n$. We construct a monotonic *abstract semantic function* $F_P : \mathcal{D}^n \to \mathcal{D}^n$ for the set of Horn clauses P, approximating the concrete semantic "immediate consequences" operator.

Since $\mathcal{D}^n$ contains infinite increasing chains, a *widening* operator for convex polyhedra [31] is needed to ensure convergence of the sequence. The sequence computed is $Z_0 = \bot^n$, $Z_{n+1} = Z_n \nabla F_P(Z_n)$ where ∇ is a widening operator for convex polyhedra and the empty polyhedron is denoted $\bot$. The conditions on ∇ ensure that the sequence stabilises; thus for some finite

j, $Z_i = Z_j$ for all $i > j$ and furthermore the value Z_j represents an over-approximation of the least model of P. Algorithm 5.4 presents convex polyhedral analysis for Horn clauses. For each constrained fact derived from the set of clauses P using this algorithm, there is a derivation tree (trace term) associated with it. These are syntactically possible trace terms using the clauses in P but may not be feasible due to abstraction. Thus all such trace terms are in $\mathcal{L}(\mathcal{A}_P)$.

Algorithm 5.4: ALGORITHM for convex polyhedral abstraction

Input: A set of CHCs P
Output: *over-approximation of the minimal model of* P

```
1  i ← 0 ;
2  Z_0 ← ⊥ ;
3  New ← ⊥ ;
4  repeat
5      foreach (p(X) ← Body) ∈ P do
6          New ← New ⊔ solve(p(X), Body, Z_i)
7      Z_{i+1} ← Z_i ∇ (New ⊔ Z_i) ;          /* Upper bound and widen */
8      i ← i + 1
9  until Z_i ⊑ Z_{i−1};
10 return Z_i;
```

Much research has been done on improving the precision of widening operators. One technique is known as widening-upto, or widening with thresholds [72]. A threshold is an assertion that is combined with a widening operator to improve its precision.

Our tool for convex polyhedral abstract interpretation, called CPA in the rest of this chapter, uses the Parma Polyhedra Library [5] to implement the operations on convex polyhedra, and incorporates a threshold generation phase based on the method described by Lakhdar-Chaouch *et al.* [108], as well as a constraint strengthening pre-processing which propagates constraints both forwards and backwards in the clauses of P.

5.4.5.1 *Computing thresholds for widening*

Recently, a technique for deriving more effective thresholds was developed [108], which we have adapted and found to be effective in experimental studies. In brief, the method collects constraints by iterating the concrete immediate consequence function S_P^D three times starting from the "top" interpretation, that is, the interpretation in which all atomic facts are true. The thresholds are computed by the following method. Let S_P^D be the standard immediate consequence operator for CHCs mentioned in Section 5.4.1. That is, if I is a set of constrained facts, $S_P^D(I)$ is the set of constrained facts that can be derived in one step from I. Given a constrained fact $p(\bar{Z}) \leftarrow \mathcal{C}$, define $\mathsf{atomconstraints}(p(\bar{Z}) \leftarrow \mathcal{C})$ to be the set of constrained facts $\{p(\bar{Z}) \leftarrow C_i \mid \mathcal{C} = C_1 \wedge \ldots \wedge C_k, (1 \leqslant i \leqslant k)\}$. The function $\mathsf{atomconstraints}$ is extended to interpretations by $\mathsf{atomconstraints}(I) = \bigcup_{p(\bar{Z}) \leftarrow \mathcal{C} \in I} \{\mathsf{atomconstraints}(p(\bar{Z}) \leftarrow \mathcal{C})\}$.

Let $I_\top$ be the interpretation consisting of the set of constrained facts $p(\bar{Z}) \leftarrow \mathit{true}$ for each predicate p. We perform three iterations of S_P^D (represented as $S_P^{D(3)}$) starting with $I_\top$ (the first three elements of a "top-down" Kleene sequence) and then extract the atomic constraints. That is, thresholds is defined as follows.

$$\mathsf{thresholds}(P) = \mathsf{atomconstraints}(S_P^{D(3)}(I_\top))$$

A difference from the method in [108] is that we use the concrete semantic function S_P^D rather than the abstract semantic function when computing thresholds. The set of threshold constraints represents an attempt to find useful predicate properties and when widening they help to preserve invariants that might otherwise be lost during widening. See [108] for further details. Threshold constraints that are not invariants are simply discarded during widening. Thresholds constraints are not necessarily over-approximations (invariants).

The operation $\mathsf{thresholds}(P)$ can become expensive and generate a very large number of constraints. Alternatively we can generate more general threshold constraints (called abstract threshold), possibly losing precision while gaining efficiency, by following more closely the approach defined in [108], using the abstract semantic function F_P. Then the threshold operation P becomes

$$\mathsf{thresholds}(P) = \mathsf{atomconstraints}(F_P^{(3)}(I_\top))$$

5.4.5.2 *Pre-processing of Horn clauses by specialisation*

The effectiveness of abstract interpretation can be improved by combining it with specialisation with respect to some property. Therefore we specialise (pre-process) the set of clauses with respect to the property to be verified using the method described in [94]. The method is summarised as follows: the inputs are a set of CHCs P and an atomic formula A (a property) and the output is P_A, a set of specialised clauses.

1. Compute a *query-answer transformation* [94] of P with respect to A, denoted P^{qa}, containing predicates p^q and p^a representing query and answer predicates for each predicate p in P.
2. Compute an over-approximation M of the model of P^{qa} using abstract interpretation.
3. Strengthen the constraints in the clauses in P, by adding constraints from the answer predicates in M. That is, for each clause

   $$p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$$

   in P, we construct a clause

   $$p(X) \leftarrow \phi, \phi_0, \phi_1, \ldots, \phi_n, p_1(X_1), \ldots, p_k(X_k)$$

   in P_A, where $p^a(X) \leftarrow \phi_0$, $p_1^a(X) \leftarrow \phi_1, \ldots, p_n^a(X) \leftarrow \phi_n$ are in M'.

The method propagates constraints globally, both forwards and backwards, and makes explicit constraints from the original clauses. This allows better analysis of the transformed clauses. Furthermore, the method is independent of the abstract domain and the constraints theory underlying the clauses.

5.5 REFINEMENT OF HORN CLAUSES USING TRACE AUTOMATA

If an over-approximation of the clauses derived by polyhedral abstraction does not contain *false*, the clauses are safe. However if *false* is contained in the approximation, we do not know whether the clauses are unsafe or whether the approximation was too imprecise. In such cases we can produce a trace term $t \in \mathcal{A}_P$ using the clauses in P which justifies the abstract derivation of *false*. The feasibility of this trace can be checked by a constraint satisfiability check. If the trace is feasible, then it corresponds to a proof of unsafety. Otherwise, refinement is considered based on this trace. In some other approaches, a more precise abstract domain is derived from the trace. In our refinement approach, which is described next, we aim to generate a modified set of clauses that could yield a better approximation. This is achieved through the steps shown in Algorithm 5.5.

Algorithm 5.5: ALGORITHM for clause refinement using FTA operations

Input: A set of Horn clauses P and an infeasible trace $t \in \mathcal{A}_P$
Output: A set of refined Horn clauses P'

1. 1. construct the trace FTA $\mathcal{A}_P$ (Definition 5.7);
2. 2. construct an FTA $\mathcal{A}_t$ such that $\mathcal{L}(\mathcal{A}_t) = \{t\}$ (Definition 5.11);
3. 3. compute the difference FTA $\mathcal{A}_P \setminus \mathcal{A}_t$ (Definition 5.5);
4. 4. generate P' from $\mathcal{A}_P \setminus \mathcal{A}_t$ and P (Algorithm 5.2) ;
5. **return** P';

Both $\mathcal{A}_P$ and $\mathcal{A}_t$ in Algorithm 5.5 are deterministic FTAs by construction, however their union is not. Determinisation is used to generate the difference FTA (step 3) and its result is in product form. The program P' has the same model (modulo predicate renaming) as P, since the steps result in the removal of an infeasible trace but all other traces are preserved.

Removal of one trace from the clauses might not seem much of a refinement. However, modifying the clauses to remove a single trace can result in significant restructuring, which arises as a side-effect of determinisation which isolates the infeasible trace. This in turn can induce a more precise abstract interpretation, with less precision loss due to convex hull operations and widening.

The correctness of this refinement follows from Proposition 5.3. In particular *false* $\in M[\![P]\!]$ if and only if *false* $\in M[\![P']\!]$ (assuming that the predicate renaming at least preserves the predicate name *false*).

Example 5.5. *Consider again the FTA shown in Example 5.4. This is in fact the determinisation of* $\mathcal{A}_P \cup \mathcal{A}_t$ *where* P *is the set of clauses in Figure 5.1 and* $\mathcal{A}_t$ *where* t *is the infeasible trace* `c3(c1)`.

The only accepting state of $\mathcal{A}_t$ is `e`; thus to construct the difference $\mathcal{A}_P \setminus \mathcal{A}_t$ we need only to remove from the automaton the states containing `e`, namely `[false,e]`. We can also remove any transitions containing this state in the right hand side. This leaves the following FTA and refined program in Figure 5.2, using the same renaming function as in Example 5.4. In this program, the infeasible trace corresponding to `c3(c1)` cannot be constructed.

```
c1 -> [mc91, e1].
c2([mc91, e1],[mc91, e1]) -> [mc91].
c2([mc91],[mc91]) -> [mc91].
c2([mc91, e1],[mc91]) -> [mc91].
c2([mc91],[mc91, e1]) -> [mc91].
c3([mc91]) -> [false].
c4([mc91]) -> [false].
c4([mc91, e1]) -> [false].
```

It can be seen that although the infeasible trace was very simple, its removal led to a considerably restructured set of clauses. We have not shown the product form here, which is in fact somewhat more compact.

The refinement process guarantees progress; that is, the infeasible computation once eliminated never arises again. Due to the construction of the id mapping for P' the traces in the languages of the FTAs of P and P' are preserved, apart from the eliminated trace.

Proposition 5.4 (Progress)**.** *Let P be a set of CHCs, and t be a trace in P. Let P' be a refined set of CHCs obtained from P after the removal of t. Then t cannot be generated from P' again.*

Proof. The proof of this proposition follows from the following points:

1. $\mathcal{A}_P$ recognizes all syntactically possible traces of P, which is an over-approximation of the traces of P since the constraints in P are not taken in account while constructing $\mathcal{A}_P$.

2. After the removal of the trace t from all possible traces of P (step 3 of Algorithm 5.5) the language of $\mathcal{A}_P \setminus \mathcal{A}_t$ does not contain t (difference automata).

3. Then using Algorithm 5.2 to generate P' from $\mathcal{A}_P \setminus \mathcal{A}_t$ and P, t will be syntactically impossible trace in P' (follows from Proposition 5.3).

4. Since t is a syntactically impossible trace in P', there is no constrained fact associated with it in this abstract domain using P' (see Section 5.4.5). □

Progress is an interesting and relevant refinement property but it gives no guarantees that a proof will eventually be found if such exists. However there exists a measure in the literature called "relative completeness", which says that the abstraction refinement procedure is complete relative to a powerful and unrealistic oracle-based method which guides the widening [7]. In the worst case the algorithm will just eliminate longer and longer infeasible traces. Even if there exists some convex polyhedral approximation that establishes P's satisfiability, the abstract interpretation algorithm involving the widening heuristic cannot guarantee to find it.

5.5.1 *Further refinement: splitting a state in the trace FTA*

We also apply a tree-automata-based transformation to split states representing predicates where convex hull operations have lost precision. A typical case is where a number of clauses with the same head predicate contain disjoint constraints, such as a predicate representing an if-then-else statement in an imperative program. The clauses defining the statement will have a clause for the *then* branch and a clause for the *else* branch. The respective constraints in these clauses are disjoint since one is the negation of the other. The convex hull will thus contain the whole space for the variables involved in these constraints.

As defined in Definition 5.6, the FTA state corresponding to such a predicate can be split. We partition the transitions corresponding to the clauses according to the disjoint groups of constraints and apply the procedure in Definition 5.6, preserving the set of traces. Thus the feasible traces and the model of the resulting clauses is preserved. This enhances precision of polyhedral analysis [59].

Splitting has to be carried out in a controlled manner to prevent blow up in the size of FTA and hence on the size of the clauses generated. With this in mind we split only those states appearing in a counterexample trace, but this is not necessary in our approach to avoid a counterexample.

It would have been possible to formulate the splitting operation directly on CHCs, without any reference to FTAs. However, since the whole procedure is based on transformations that preserve the set of feasible traces, we preferred to present splitting as a language-preserving operation on FTAs.

5.6 EXPERIMENTS ON CHC BENCHMARK PROBLEMS

5.6.1 *Experimental settings*

Our tool consists of an implementation of a *convex polyhedra analyser* for CLP written in Ciao Prolog[1] interfaced to the Parma Polyhedra Library [5] as well as an implementation of an FTA determiniser written in Java. It takes as input a CLP program and returns "safe", "unsafe" or "unknown" (after resources are exhausted). The input is first pre-processed using the method described in 5.4.5.2. The benchmark set contains 216 CHCs verification problems (179 safe and 37 unsafe problems), taken mainly from the repositories of several state-of-the-art software verification tools such as DAGGER [67] (21 problems), TRACER [85] (66 problems), InvGen [68] (68 problems), and also from the TACAS 2013 Software Verification Competition [12] (52 problems). These problems are also available in `C` (`http://akira.ruc.dk/~kafle/comlan-vmcai15-benchmarks.zip`) and they were first translated to CLP form[2]. The chosen problems are representatives of different categories of the Software Verification Competition (loops, control flow and integer, SystemC etc.) as well as specific problems used to demonstrate the strength of different verification tools. The benchmarks in CLP form are available

1 http://ciao-lang.org/
2 Thanks to Emanuele De Angelis for the translation

Result	CPA	CPA+R	CPA+R+Split	QARMC	VeriMAP (GenPH)	TRACER-SPost	TRACER-WPre	ELDARICA
solved (safe/unsafe)	160 (142/18)	182 (160/22)	195 (164/31)	178 (141/37)	185 (154/31)	91 (74/17)	103 (85/18)	159(135/24)
timeout or errors	56	34	22	38	38	125	113	57
average time (secs.)	5.98	51.66	50.08	59.1	57.93	305.03	225.82	50.02

Figure 5.4: Experimental results on 216 (179 safe / 37 unsafe) CHC verification problems with a timeout of five minutes

from `http://akira.ruc.dk/~kafle/VMCAI15-Benchmarks.zip`. The experiments were carried out on an Intel(R) computer with a 2.66GHz processor running Debian 5 in 6 GB memory.

5.6.2 *Summary of results*

The results of our experiments are summarised in Table 5.4. Column CPA summarises the results using our own *convex polyhedra analyser* (Section 5.4.5) with no refinement step. Column CPA+R shows the results obtained by iterating the CPA algorithm with the refinement step described in Section 5.5, Algorithm 5.5. Column CPA+R+Split incorporates the FTA-based state splitting into the refinement step (Section 5.5.1). In all the above cases, we used a concrete threshold generation as described in 5.4.5.1. Column QARMC shows the results obtained on the same problems using the QARMC tool [134, 64]. The columns VeriMAP(GenPH), TRACER-SPost, TRACER-WPre respectively report results using the VeriMAP system implementing Iterated Specialization method with the generalization operator GenPH [38], TRACER [85] using the strongest postcondition (SPost) and the weakest precondition (WPre) options. The results in these three columns are taken from [38] since we couldn't run these tools. We used the same set of benchmarks as in [38]. The last column ELDARICA reports results using Eldarica tool [80] which uses disjunctive interpolants for the Horn clause verification purpose [137].

5.6.3 *Discussion of results*

The results show that CPA is reasonably effective on its own, solving 74% (160/216) of the problems. When combined with a refinement phase we can solve further 22 problems. Although only one infeasible trace is eliminated in each refinement step, the refined program splits some of the predicates appearing in the trace, which we noted to be a crucial point of precision for polyhedral analysis [59]. When adding the state splitting refinement we solve an additional 13 problems. Further splitting would solve more problems but we are unwilling to introduce uncontrolled splitting due to the blow up in program size that could result. The maximum number of iterations required to solve a problem was 8. Although the timeout limit was five minutes, only 5% of the solved problems required more than one minute.

Our implementation uses the product form for DFTAs produced by the determinisation algorithm, although the formalisation of refinement in Section 5.5 uses only standard FTA transitions. Although the traces for clauses with predicates produced from product states differ from the original clauses, they can be regarded as representing the original traces, by

unfolding the clauses resulting from ϵ-transitions. Product form adds to the scalability of the approach, especially for Horn clauses with more than one body atom. Empirically we have not shown this here but this is due to the scalability of the product form during determinisation of FTAs (see [60]).

5.6.4 *Comparison with other tools*

On the set of verification problems considered, our results (*CPA+R+Split*) improve on other tools both in average time and the number of instances solved. The results of VeriMAP and QARMC are close to ours while results of TRACER is bit far. This is due to the fact that TRACER uses symbolic execution and does not scale well. Out of 216 problems QARMC solves 178 problems with an average time of 59 seconds whereas we can solve 195 problems with an average time of 50 seconds. However, all unsafe programs in the benchmark set are solved by QARMC in contrast to ours. Surprisingly enough, the number of unsafe problems solved by VeriMAP and our tool is the same. The model checking algorithm implemented in Eldarica for Horn solving is similar in spirit as the one described in [64] but uses disjunctive interpolation for counterexamples generalization instead of tree interpolation which is strictly more general than tree interpolation [137]. We suspect that it is due to this, the average time taken by Eldarica is slightly less than that of QARMC though it solves a lesser number of instance than QARMC. Our results show that for these set of examples, tools using polyhedral abstraction seems more powerful than the others.

Convex polyhedral analysis is good at finding the required invariants to prove the safety of a program and due to this our tool and VeriMAP solved more safe problems than QARMC. On the other hand, QARMC seems to be more effective at finding bugs. Most of the problems challenging to us come from some particular categories e.g. SystemC (modeled over fixed size integers) and Control Flow and Integer Variables of [12] which requires some specific techniques to solve. Safe problems challenging to us are also challenging to QARMC though this is not the case for unsafe problems.

5.6.5 *Additional experiments on SV-COMP-15*

We chose a subset of 132 problems from SV-COMP 2015[3] [13]. This set contains benchmarks from the categories which were not reported in our experiments before such as *recursive benchmarks* which needs recursive analysis. Additionally it contains some benchmarks from *Loop* category such as *loop-acceleration*, *loop-lit*, *loop-new*. We used SeaHorn [71, 70], a verification framework based on LLVM, for Horn clause generation. SeaHorn first compiles C to LLVM intermediate representation (LLVM IR), also known as bitcode using *clang*, a C-family front-end for LLVM[4]. The bitcode is further simplified and optimized reusing the vast amount of work done on LLVM (e.g. function inlining, dead code elimination, CFG simplifications etc.) whose purpose is to make the verification task easier. The resulting bitcode is translated to

3 http://sv-comp.sosy-lab.org/2015/benchmarks.php
4 http://clang.llvm.org/

Result	CPA+R+Split	CPA+R+Split+AT	QARMC
solved (safe/unsafe)	114 (54/60)	114 (54/60)	110 (42/68)
timeout or errors	18	18	22
average time (secs.)	72.92	66.28	52.4

Table 5.1: Experimental results on 132 CHC verification problems with a timeout of five minutes

Horn clauses using different semantics for example small step, large block encoding etc. More details can be found in [71, 70]. These benchmarks are available in C as well as in Horn clause form from `http://akira.ruc.dk/~kafle/comlan-vmcai15-benchmarks.zip`. The results are summarised in the Table 5.1. The column *CPA+R+Split+AT* reports results using our tool described above which now uses abstract threshold as described in 5.4.5.1 rather than the concrete one. The results show that our tool with the option abstract threshold (column 2) scales more than the concrete one (column 1) though the number of instances solved are the same. In these benchmarks, though we solve a few more problems than QARMC, it is much faster than our tool.

5.7 RELATED WORK

The work by Heizmann et al. [74, 76] uses nested word automata to construct a framework for abstraction refinement. Our work could certainly be regarded as extending that framework to tree-structured computations, using tree automata instead of (nested) word automata. However our aim is somewhat different. We use automata techniques to *perform* the refinement whereas in [74] automata notation is only used to re-express the verification problem, shifting the verification problem to the construction of “interpolant automata", without providing any automata-based algorithms to do this. On the other hand we discuss the practicality of the automata-based approach on a set of challenging problems.

While we eliminate only one trace at a time in the described procedure, the FTA difference algorithm extends naturally to eliminating (infinite) sets of traces. This is a goal that is well worth pursuing – although to find an interpolant automaton describing an infinite set of infeasible traces is sometimes as difficult as solving the original problem.

Verification of CLP programs using abstract interpretation and specialisation has been studied for some time. The use of an over-approximation of the semantics of a program can be used to establish safety properties – if a state or property does not appear in an over-approximation, it certainly does not appear in the actual program behaviour. A general framework for logic program verification through abstraction was described by Levi [117]. Peralta *et al.* [129] introduced the idea of using a Horn clause representation of imperative languages and a convex polyhedral analyser to discover invariants of a program. Another approach is taken in the work of De Angelis *et al.* [38, 37] on applying program specialisation to achieve verification. Unfolding and folding operations play a vital role in that approach, and hence the program structure is changed much more fundamentally than in our approach.

CEGAR [23] has been successfully used in verification to automatically refine (predicate) abstractions [21, 109] to reduce false alarms but not much has been explored in refining abstractions in the convex polyhedral domain. Some work on this (with progress guarantee) has been done in [4] and [67]. [4] uses the powerset domain, while [67] uses a Hint DAG to gain precision lost during the convex hull operation. Both make use of interpolation. The use of interpolation in refinement in verification of Horn clauses is explored in [15, 69]. In our approach we guarantee elimination of only one trace and elimination of others depends on properties of the abstract interpretation techniques. One drawback of our approach is that we cannot characterize what other infeasible traces are removed by the refinement if there is any. By contrast in interpolation-based techniques the refinement introduces new properties which guarantee progress and the elimination of all counterexamples covered by those properties. However the effectiveness of interpolation-based refinement depends on the generation of "good" interpolants, which is a matter of continuing research, for example by Rümmer *et al.* [137]. There is no generalisation of counterexamples currently since we remove a single counterexample in each iteration. In this sense our refinement approach is weak. The ideas developed in [137] are certainly useful to extend our refinement approach. A number of tools implementing predicate abstraction and refinement are available, such as HSF [64] and BLAST [8]. TRACER [62] is a verification tool based on CLP that uses symbolic execution.

A point of contrast is that in our approach, the refinements are embedded in the clauses whereas in CEGAR they are accumulated in the set of properties used for property-based abstraction. Also we rely on the abstraction using convex polyhedral analysis to discover invariants whereas CEGAR-based approaches rely on interpolation in the refinement stage to perform generalisation, thus discovering useful properties. A weakness of invariant generation using interpolation is that the interpolants must share variables with the unsatisfiable part of the constraints, typically those in the integrity constraints, which can be insufficient for finding invariants of inner recursive predicates. Polyhedral analysis is more expensive, yet seems (along with the threshold assertions, see Section 5.4.5) to be very effective at finding invariants even on the first iteration.

Informally one can say that approaches differ in where the "hard work" is performed. In the CEGAR approaches and in [74] the refinement step is crucial, and interpolation plays a central role. In our approach, by contrast, most of the hard work is done by the abstract interpretation, which finds useful invariants. Finding the most effective balance between abstraction and refinement techniques is a matter of ongoing research.

5.8 CONCLUSION AND FUTURE WORK

We presented a procedure for abstraction refinement in Horn clause verification based on tree automata. This was achieved through a combination of abstraction (using abstraction interpretation) followed by a trace refinement (using finite tree automata). The refinement is independent of the abstract domain used. The practicality of our approach was demonstrated on a set of Horn clause verification problems.

In the future, we will investigate the elimination of a larger set of infeasible traces in each refinement step, possibly by generalising a trace using interpolation or by discovering a set

of infeasible traces. At the moment, a new program is generated after refinement and the analysis is restarted from the scratch. In the future, we would like to reuse the result of analysis from the previous iterations and build on this instead of starting the analysis from the scratch. The optimisation of our tool chain is also an important topic for future work as it is clear that our prototype, built by chaining together tools using shell scripts, contains much redundancy.

6

INTERPOLANT TREE AUTOMATA AND THEIR APPLICATION IN HORN CLAUSE VERIFICATION

With John P. Gallagher

Abstract
This chapter investigates the combination of abstract interpretation over the domain of convex polyhedra with interpolant tree automata, in an abstraction-refinement scheme for Horn clause verification. These techniques have been previously applied separately, but are combined in a new way in this chapter. The role of an interpolant tree automaton is to provide a generalisation of a spurious counterexample during refinement, capturing a possibly infinite set of spurious counterexample traces. In our approach these traces are then eliminated using a transformation of the Horn clauses. We compare this approach with two other methods; one of them uses interpolant tree automata in an algorithm for trace abstraction and refinement, while the other uses abstract interpretation over the domain of convex polyhedra without the generalisation step. Evaluation of the results of experiments on a number of Horn clause verification problems indicates that the combination of interpolant tree automaton with abstract interpretation gives some increase in the power of the verification tool, while sometimes incurring a performance overhead.

Keywords: Interpolant tree automata, Horn clauses, Abstraction-refinement, Horn clause verification.

6.1 INTRODUCTION

In this chapter we combine two existing techniques, namely abstract interpretation over the domain of convex polyhedra and interpolant tree automata in a new way for Horn clause verification. Abstract interpretation is a scalable program analysis technique which computes invariants allowing many program properties to be proven, but suffers from false alarms; safe but not provably safe programs may be indistinguishable from unsafe programs. Refinement is considered in this case. In previous work [96] we described an *abstraction-refinement* scheme for Horn clause verification using abstract interpretation and refinement with finite tree automata. In that approach refinement eliminates a single spurious counterexample in each iteration of the abstraction-refinement loop, using a clause transformation based on a tree automata difference operation. In contrast to that work, we apply the method of Wang and Jiao [147] for constructing an *interpolant tree automaton* from an infeasible trace. This generalises the trace of a spurious counterexample, recognising a possibly infinite number of spurious counterexamples, which can then be eliminated in one iteration of the abstraction-refinement loop. We combine this construction in the framework of [96]. The experimental results on a

```
c1. fib(A, B):- A>=0, A=<1, B=1.
c2. fib(A, B) :- A>1, A2=A-2,
           A1=A-1, B=B1+B2,  fib(A1,B1), fib(A2,B2).
c3. false:- A>5, B<A, fib(A,B).
```

Figure 6.1: Example CHCs (Fib) defining a Fibonacci function.

set of Horn clause verification problems are reported, and compared with both [96] and the results of Wang and Jiao [147] using trace abstraction and refinement.

In Section 6.2 we introduce the key concepts of constrained Horn clauses and finite tree automata. Section 6.3 contains the definitions of interpolants and the construction of an interpolant tree automaton following the techniques of Wang and Jiao [147]. In Section 6.4 we describe our algorithm combining abstract interpretation with interpolant tree automata, including in Section 6.4.1 an experimental evaluation and comparison with other approaches. Finally in Section 6.5 we discuss related work.

6.2 PRELIMINARIES

A constrained Horn clause (CHC) is a first order predicate logic formula of the form $\forall(\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k) \rightarrow p(X))$ $(k \geqslant 0)$, where ϕ is a first order logic formula (constraint) with respect to some background theory and $p_1, \ldots, p_k, p$ are predicate symbols. We assume (wlog) that X_i, X are (possibly empty) tuples of distinct variables and ϕ is expressed in terms of X_i, X, which can be achieved by adding equalities to ϕ. $p(X)$ is the head of the clause and $\phi \wedge p_1(X_1) \wedge \ldots \wedge p_k(X_k)$ is the body. There is a distinguished predicate symbol *false* which is interpreted as FALSE. Clauses whose head is *false* are called *integrity constraints*. Following the notation used in constraint logic programming a clause is usually written as $H \leftarrow \phi, B_1, \ldots, B_k$ where $H, B_1, ..., B_k$ stand for atomic formulas (atoms) $p(X), p_1(X_1), ..., p_k(X_k)$. A set of CHCs is sometimes called a (constraint logic) program.

An interpretation of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \phi$ where A is an atom and ϕ is a formula with respect to some background theory. $A \leftarrow \phi$ represents a set of ground facts $A\theta$ such that $\phi\theta$ holds in the background theory (θ is called a grounding substitution). An interpretation that satisfies each clause in P is called a model of P. In some works [15, 126], a *model* is also called a *solution* and we use them interchangeably in this chapter.

HORN CLAUSE VERIFICATION PROBLEM. Given a set of CHCs P, the CHC verification problem is to check whether there exists a model of P. It can easily be shown that P has a model if and only if the fact *false* is not a consequence of P.

An example set of CHCs, encoding the Fibonacci function is shown in Figure 6.1. Since its derivations are trees, it serves as an interesting example from the point of view of *interpolant tree automata*.

Definition 6.1 (Finite tree automaton [28]). *An FTA $\mathcal{A}$ is a tuple (Q, Q_f, Σ, Δ), where Q is a finite set of states, $Q_f \subseteq Q$ is a set of final states, Σ is a set of function symbols, and Δ is a set of transitions*

of the form $f(q_1, \ldots, q_n) \to q$ *with* $q, q_1, \ldots, q_n \in Q$ *and* $f \in \Sigma$. *We assume that* Q *and* Σ *are disjoint.*

We assume that each CHC in a program P is associated with an identifier by a mapping $\mathrm{id}_P : P \to \Sigma$. An identifier (an element of Σ) is a function symbol whose arity is the same as the number of atoms in the clause body. For instance a clause $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k)$ is assigned a function symbol with arity k. As will be seen later, these identifiers are used to build trees that represent *derivations* using the clauses. A set of derivation trees (traces) of a set of atoms of a program P can be abstracted and represented by an FTA. We provide such a construction in Definition 6.2.

Definition 6.2 (Trace FTA for a set of CHCs)**.** *Let* P *be a set of CHCs. Define the trace FTA for* P *as* $\mathcal{A}_P = (Q, Q_f, \Sigma, \Delta)$ *where*

- $Q = \{p \mid p \text{ is a predicate symbol of } P\} \cup \{\mathit{false}\}$;
- $Q_f = \{\mathit{false}\}$;
- Σ is a set of function symbols;
- $\Delta = \{c_j(p_1, \ldots, p_k) \to p \mid \text{where } c_j \in \Sigma,\ p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k) \in P,\ c_j = \mathrm{id}_P(p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k))\}$.

The elements of $\mathcal{L}(\mathcal{A}_P)$ are called trace-terms or trace-trees or simply traces of P rooted at *false*.

Example 6.1. *Let* Fib *be the set of CHCs in Figure 6.1. Let* $\mathrm{id}_{\mathtt{Fib}}$ *map the clauses to identifiers* c_1, c_2, c_3 *respectively. Then* $\mathcal{A}_{\mathtt{Fib}} = (Q, Q_f, \Sigma, \Delta)$ *where:*

$$
\begin{aligned}
Q &= \{\mathtt{fib}, \mathtt{false}\} \\
Q_f &= \{\mathtt{false}\} \\
\Sigma &= \{c_1, c_2, c_3\} \\
\Delta &= \{c_1 \to \mathtt{fib},\ c_2(\mathtt{fib}, \mathtt{fib}) \to \mathtt{fib}, \\
&\qquad c_3(\mathtt{fib}) \to \mathtt{false}\}
\end{aligned}
$$

Similarly, we can also construct an FTA representing a single trace. It should be noted that the whole idea of representing program traces by FTAs is to use automata theoretic operations for dealing with program traces, for example, removal of an undesirable trace from a set of program traces. Let P be a set of CHCs and let $t \in \mathcal{L}(\mathcal{A}_P)$. There exists an FTA $\mathcal{A}_t$ such that $\mathcal{L}(\mathcal{A}_t) = \{t\}$. We illustrate the construction with an example.

Example 6.2 (Trace FTA)**.** *Consider the FTA in Example 6.1. Let* $t = c_3(c_2(c_1, c_1)) \in \mathcal{L}(\mathcal{A}_P)$. *Then* $\mathcal{A}_t = (Q, Q_f, \Sigma, \Delta)$ *is defined as:*

$$\begin{aligned} Q &= \{e_1, e_2, e_3, e_4\} \\ Q_f &= \{e_1\} \\ \Sigma &= \{c_1, c_2, c_3, c_4\} \\ \Delta &= \{c_1 \to e_3,\ c_1 \to e_4,\ c_2(e_3, e_4) \to e_2, \\ &\quad\ c_3(e_2) \to e_1\} \end{aligned}$$

where Σ is the same as in $\mathcal{A}_P$ and the states e_i $(i = 1 \ldots 4)$ represent the nodes in the trace-tree, with root node e_1 as the final state.

A trace-term is a representation of a *derivation trees*, also called an AND-tree [142, 56] giving a proof of an atomic formula from a set of CHCs.

Definition 6.3 (AND-tree for a trace term $T(t)$ (adapted from [96]))**.** *Let P be a set of CHCs and let $t \in \mathcal{L}(\mathcal{A}_P)$. An AND-tree corresponding to t, denote by $T(t)$, is the following labelled tree, where each node of $T(t)$ is labelled by an atom, a clause identifier and a constraint.*

1. *For each sub-term $c_j(t_1, \ldots, t_k)$ of t there is a corresponding node in $T(t)$ labelled by an atom $p(X)$, an identifier c_j such that $c_j = id_P(p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k))$, and a constraint ϕ; the node's children (if $k > 0$) are the nodes corresponding to $t_1, \ldots, t_k$ and are labelled by $p_1(X_1), \ldots, p_k(X_k)$.*
2. *The variables in the labels are chosen such that if a node n is labelled by a clause, the local variables in the clause body do not occur outside the subtree rooted at n.*

We assume that each node in $T(t)$ is uniquely identified by a natural number. We omit t from $T(t)$ when it is clear from the context.

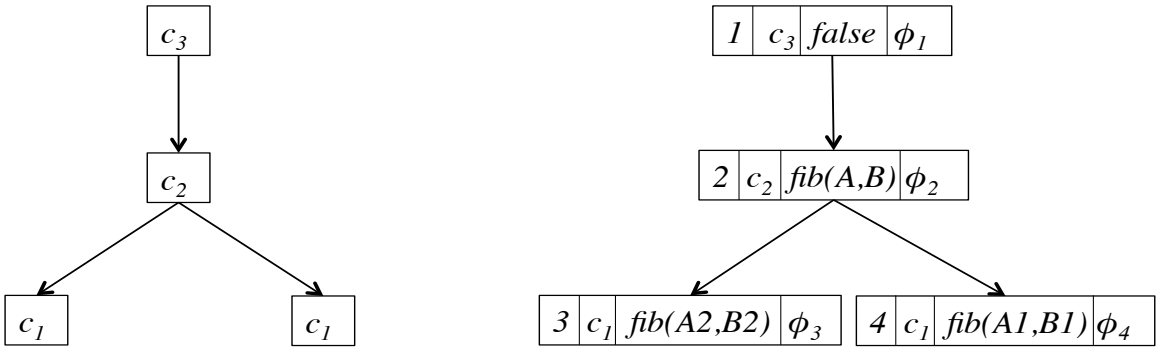

Figure 6.2: A trace-term $c_3(c_2(c_1, c_1))$ of Fib (left) and its AND-tree (right), where $\phi_1 \equiv A > 5 \wedge B < A$; $\phi_2 \equiv A > 1 \wedge A2 = A - 2 \wedge A1 = A - 1 \wedge B = B1 + B2$; $\phi_3 \equiv A2 \geqslant 0 \wedge A2 \leqslant 1 \wedge B2 = 1$; $\phi_4 \equiv A1 \geqslant 0 \wedge A1 \leqslant 1 \wedge B1 = 1$.

The formula represented by an AND-tree T, denoted by $F(T)$ is

1. ϕ, if T is a single leaf node labelled by a constraint ϕ; or
2. $\phi \wedge \bigwedge_{i=1..n}(F(T_i))$ if the root node of T is labelled by the constraint ϕ and has subtrees $T_1, \ldots, T_n$.

The formula $F(T)$ where T is the AND-tree in Figure 6.2 is

$$A > 5 \wedge B < A \wedge A > 1 \wedge A2 = A - 2 \wedge A1 = A - 1 \wedge B = B1 + B2$$
$$A2 \geqslant 0 \wedge A2 \leqslant 1 \wedge B2 = 1 \wedge A1 \geqslant 0 \wedge A1 \leqslant 1 \wedge B1 = 1$$

We say that an AND-tree T is satisfiable or feasible if $F(T)$ is satisfiable, otherwise unsatisfiable or infeasible. Similarly, we say a trace-term is satisfiable (unsatisfiable) iff its corresponding AND-tree is satisfiable (unsatisfiable). The trace-term $c_3(c_2(c_1, c_1))$ in Figure 6.2 is unsatisfiable since $F(c_3(c_2(c_1, c_1)))$ is unsatisfiable.

6.3 INTERPOLANT TREE AUTOMATA

Refinement of trace abstraction is an approach to program verification [74]. In this approach, if a property is not provable in an abstraction of program traces then an abstract trace showing the violation of the property is emitted. If such a trace is not feasible with respect to the original program, it is eliminated from the trace abstraction which is viewed as a refinement of the trace abstraction. The notion of interpolant automata [74] allows one to generalise an infeasible trace to capture possibly infinitely many infeasible traces which can then be eliminated in one refinement step. In this section, we revisit the construction of an *interpolant tree automaton* [147] from an infeasible trace-tree. The automaton serves as a generalisation of the trace-tree; and we apply this construction in Horn clause verification.

Definition 6.4 ((Craig) Interpolant [35])**.** *Given two formulas ϕ_1, ϕ_2 such that $\phi_1 \wedge \phi_2$ is unsatisfiable, a (Craig) interpolant is a formula I with (1) $\phi_1 \rightarrow I$; (2) $I \wedge \phi_2 \rightarrow$ False; and (3) vars$(I) \subseteq$ vars$(\phi_1) \cap$ vars(ϕ_2). An interpolant of ϕ_1 and ϕ_2 is represented by $I(\phi_1, \phi_2)$.*

The existence of an interpolant implies that $\phi_1 \wedge \phi_2$ is unsatisfiable [136]. Similarly, if the background theory underlying the CHCs P admits (Craig) interpolation [35], then every infeasible derivation using the clauses in P has an interpolant [126].

Example 6.3 (Interpolant example)**.** *Let $\phi_1 \equiv A2 \leqslant 1 \wedge A > 1 \wedge A2 = A - 2 \wedge A1 = A - 1 \wedge B = B1 + B2$ and $\phi_2 \equiv A > 5 \wedge B < A$ such that $\phi_1 \wedge \phi_2$ is unsatisfiable. Since the formula $I \equiv A \leqslant 3$ fulfills all the conditions of Definition 6.4, it is an interpolant of ϕ_1 and ϕ_2.*

Given a node i in an AND-tree T, we call T_i the sub-tree rooted at i, ϕ_i the formula label of node i, $F(T_i)$ the formula of the sub-tree rooted at node i, and $G(T_i)$ the formula $F(T)$ except the formula $F(T_i)$, which is defined as follows:

1. *true*, if T is a single leaf node labelled by constraint ϕ and the node is i; or
2. ϕ, if T is a single leaf node labelled by constraint ϕ and the node is different from i; or

3. *true*, if the root node of T is labelled by the constraint ϕ and the node is i; or
4. $\phi \wedge \bigwedge_{l=1..n} G(T_l)$ if the root node of T is labelled by the constraint ϕ, and the node is different from i and has subtrees $T_1, \ldots, T_n$.

For example in the example program of Figure 6.2, $G(T_1) = \mathit{true}$ and $G(T_2) = \phi_1$.

Definition 6.5 (Tree Interpolant of an AND-tree [18]). *Let T be an infeasible* AND*-tree. A tree interpolant $TI(T)$ for T is a tree constructed as follows:*

1. *The root node i of $TI(T)$ is labelled by i, the atom of the node i of T and the formula false;*
2. *Each leaf node i of $TI(T)$ is labelled by i, the atom of the node i of T and by $I(F(T_i), G(T_i))$;*
3. *Let i be any other node of T. We define F_1 as $(\phi_i \wedge \bigwedge_{k=1}^{n} I_k)$ where $\bigwedge_{k=1}^{n} I_k$ $(n \geqslant 1)$ is the conjunction of formulas representing the interpolants of the children of the node i in $TI(T)$. Then the node i of $TI(T)$ is labelled by i, the atom of the node i of T and the formula $I(F_1, G(T_i))$.*

Note that the formula F_1 and $G(T_i)$ in Definition 6.5 (point 3) is unsatisfiable becuase of the property of interpolant and the fact that $F(T)$ is unsatisfiable. The *tree interpolant* corresponding to *AND tree* in Figure 6.2(b) is shown in Figure 6.3(b).

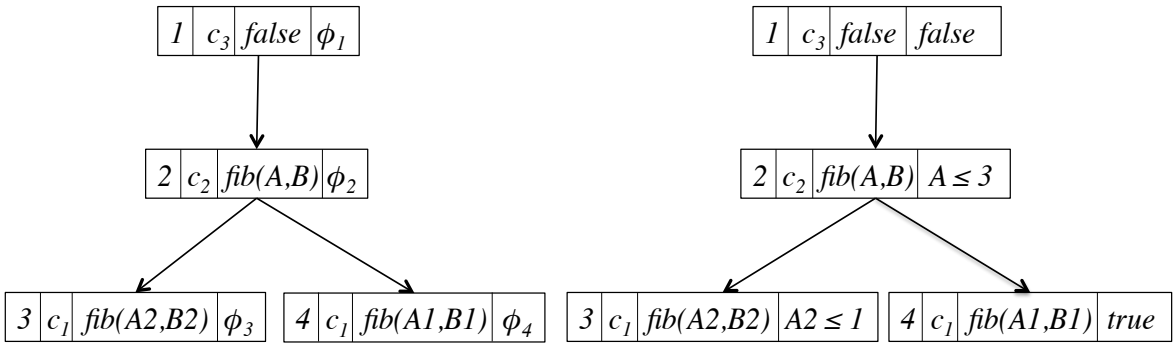

Figure 6.3: *AND tree* of Figure 6.2 (left) and its *tree interpolant* (right). Let I_j represents an interpolant of the node j. Then $I_1 \equiv \mathit{false}$; $I_4 \equiv I(\phi_4, \phi_3 \wedge \phi_1 \wedge \phi_2)$; $I_3 \equiv I(\phi_3, \phi_1 \wedge \phi_2 \wedge I_4)$; $I_2 \equiv I(I_3 \wedge I_4 \wedge \phi_2, \phi_1)$.

Since there is a one-one correspondence between an AND-tree and a trace-term, we can define a tree interpolant for a trace-term as follows:

Definition 6.6 (Tree Interpolant of a trace-term $TI(t)$). *Given an infeasible trace-term t, its tree interpolant, represented as $TI(t)$, is the tree interpolant of its corresponding* AND*-tree.*

Definition 6.7 (Interpolant mapping Π_{TI}). *Given a tree interpolant TI for some tree, Π_{TI} is a mapping from the atom labels and node numbers of each node in TI to the formula label such that $\Pi_{TI}(A^j) = \psi$ where A is the atom label and ψ is the formula label at node j.*

For our example program Π_{TI} is the following:

$$\{\mathtt{false}^1 \mapsto \mathit{false}, \mathtt{fib}^2(A, B) \mapsto A \leqslant 3, \mathtt{fib}^3(A2, B2) \mapsto A \leqslant 1, \mathtt{fib}^4(A1, B1) \mapsto \mathit{true}\}$$

Property 6.1 (Tree interpolant property). *Let* $\mathsf{TI}(\mathsf{T})$ *be a tree interpolant for some infeasible* AND*-tree* T*. Then*

1. $\Pi_{\mathsf{TI}}(r^i) =$ *false where* r *is the atom label of the root of* $\mathsf{TI}(\mathsf{T})$*;*
2. *for each node* j *with children* $j_1, ..., j_n$ $(n \geqslant 0)$ *the following property holds:* $(\bigwedge_{k=0}^{n} \Pi_{\mathsf{TI}}(A^{j_k})) \wedge \phi_j \rightarrow \Pi_{\mathsf{TI}}(A^j)$ *where* ϕ_j *is the formula label of the node* j *of* T*;*
3. *for each node* j *the following property holds:* $vars(\Pi_{\mathsf{TI}}(A^j)) \subseteq (vars(F(T_j)) \cap vars(G(T_j)))$*, where the formula* $F(T_j)$ *and* $G(T_j)$ *corresponds to* T*.*

Definition 6.8 (Interpolant tree automaton for Horn clauses $\mathcal{A}_t^I$=(Q, Q_f, Σ, Δ) [147]). *Let* P *be a set of CHCs,* $t \in \mathcal{L}(\mathcal{A}_P)$ *be any infeasible trace-term and* $\mathsf{TI}(t)$ *be a tree interpolant of* t*. Let* $\sigma : As \times J \rightarrow Q$ *where* σ *maps an atom at node* $i \in J$ *of* $\mathsf{TI}(t)$ *to an FTA state in* Q*. Define* $\rho : Pred^J \rightarrow Pred$ *which maps a predicate name with superscript to a predicate name of* P*. Then the interpolant automaton of* t *is defined as an FTA* $\mathcal{A}_t^I$ *such that*

- $Q = \{\sigma(A, i) : A \text{ is the atom label of the node } i \text{ of } \mathsf{TI}(t)\}$;
- $F = \{\sigma(A, i) : A \text{ is the atom label of the root of } \mathsf{TI}(t)\}$;
- Σ *is a set of function symbols of* P*;*
- $\Delta = \{c(p_1^{j_1}, \ldots, p_k^{j_k}) \rightarrow p^j \mid cl = p(X) \leftarrow \phi, p_1(X_1), \ldots, p_k(X_k) \in P,\ c = id_P(cl),\ \rho(p^i) = p\ ,\ \rho(p_m^i) = p_m \text{ for } m = 1..k \text{ and } \Pi_{\mathsf{TI}}(p^j)(X) \leftarrow \phi, \Pi_{\mathsf{TI}}(p_1^{j_1})(X_1), \ldots, \Pi_{\mathsf{TI}}(p_k^{j_k})(X_k)\}$.

Example 6.4 (Interpolant automata for $c_3(c_2(c_1, c_1))$).

$$
\begin{aligned}
Q &= \{\mathtt{fib}^2, \mathtt{fib}^3, \mathtt{fib}^4, \mathtt{error}\} \\
Q_f &= \{\mathtt{error}\} \\
\Sigma &= \{c_1, c_2, c_3\} \\
\Delta &= \{c_1 \rightarrow \mathtt{fib}^2, c_1 \rightarrow \mathtt{fib}^3, c_1 \rightarrow \mathtt{fib}^4, \\
&\quad c_2(\mathtt{fib}^2, \mathtt{fib}^2) \rightarrow \mathtt{fib}^4, c_2(\mathtt{fib}^2, \mathtt{fib}^3) \rightarrow \mathtt{fib}^2, \\
&\quad c_2(\mathtt{fib}^2, \mathtt{fib}^3) \rightarrow \mathtt{fib}^4, c_2(\mathtt{fib}^2, \mathtt{fib}^4) \rightarrow \mathtt{fib}^4, \\
&\quad c_2(\mathtt{fib}^3, \mathtt{fib}^2) \rightarrow \mathtt{fib}^2, c_2(\mathtt{fib}^3, \mathtt{fib}^2) \rightarrow \mathtt{fib}^4, \\
&\quad c_2(\mathtt{fib}^3, \mathtt{fib}^3) \rightarrow \mathtt{fib}^2, c_2(\mathtt{fib}^3, \mathtt{fib}^3) \rightarrow \mathtt{fib}^4, \\
&\quad c_2(\mathtt{fib}^3, \mathtt{fib}^4) \rightarrow \mathtt{fib}^2, c_2(\mathtt{fib}^3, \mathtt{fib}^4) \rightarrow \mathtt{fib}^4, \\
&\quad c_2(\mathtt{fib}^4, \mathtt{fib}^2) \rightarrow \mathtt{fib}^4, c_2(\mathtt{fib}^4, \mathtt{fib}^3) \rightarrow \mathtt{fib}^2, \\
&\quad c_2(\mathtt{fib}^4, \mathtt{fib}^3) \rightarrow \mathtt{fib}^4, c_2(\mathtt{fib}^4, \mathtt{fib}^4) \rightarrow \mathtt{fib}^4, \\
&\quad c_3(\mathtt{fib}^2) \rightarrow \mathtt{error}, c_3(\mathtt{fib}^3) \rightarrow \mathtt{error}\}
\end{aligned}
$$

The construction described in Definition 6.8 recognizes only infeasible traces terms of P as stated in Theorem 6.1.

Theorem 6.1 (Soundness). *Let* P *be a set of CHCs and* $t \in \mathcal{L}(\mathcal{A}_P)$ *be any infeasible trace-term. Then the interpolant automaton* $\mathcal{A}^I_t$ *recognises only infeasible trace-terms of* P.

It is hard to characterise the set of traces. But these are the traces which share the same reason of unsatisfiability as t.

Definition 6.9 (Conjunctive interpolant mapping). *Given an interpolant mapping* Π_{TI} *of a tree interpolant* TI, *we define a conjunctive interpolant mapping for an atom label* A *of any node in* TI, *represented as* $\Pi^c_{TI}(A)$, *to be the following formula* $\Pi^c_{TI}(A) = \bigwedge_j \Pi_{TI}(A^j)$, *where* j *ranges over the nodes of* TI. *It is the conjunction of interpolants of all the nodes of* TI *with atom label* A. *The conjunctive interpolant mapping of* TI *is represented is* $\Pi^c_{TI} = \{\Pi^c_{TI}(A) \mid A \text{ is the atom label of TI}\}$.

It is desirable that the *interpolant tree automaton* of a trace $t \in \mathcal{L}(\mathcal{A}_P)$ recognizes as many infeasible traces as possible, in an ideal situation, all infeasible traces of P. This is possible under the condition described in Theorem 6.2.

Theorem 6.2 (Model and Interpolant Automata). *Let* $t \in \mathcal{L}(\mathcal{A}_P)$ *be any infeasible trace-term. If* $\Pi^c_{TI(t)}$ *is a model of* P, *then the interpolant automaton of* t *recognises all infeasible trace-terms of* P.

6.4 APPLICATION TO HORN CLAUSE VERIFICATION

An *abstraction-refinement* scheme for Horn clause verification is described in [96] which is depicted in Figure 6.4. In this, a set of CHCs P is analysed using the techniques of *abstract interpretation* over the domain of convex polyhedra which produces an over-approximation M of the minimal model of P. The set of traces used during the analysis can be captured by an FTA $\mathcal{A}^M_P$ (see Definition 6.10). This automaton recognizes all trace-terms of P except some infeasible ones. Some of the infeasible trace-terms are removed by the abstract interpretation. P is solved or safe (that is, it has a model) if $\mathit{false} \notin M$. If this is not the case, a trace-term $t \in \mathcal{L}(A^M_P)$ is selected and checked for feasibility. If the answer is positive, P has no model, that is, P is unsafe.

Otherwise t is considered spurious and this drives the refinement process. The refinement in [96] consists of constructing an automaton $\mathcal{A}'_P$ which recognizes all traces in $\mathcal{L}(\mathcal{A}^M_P) \setminus \mathcal{L}(\mathcal{A}_t)$ and generating a refined set of clauses from P and $\mathcal{A}'_P$. The automata difference construction refines a set of traces (abstraction), which induces refinement in the original program. The refined program is again fed to the abstract interpreter. This process continues until the problem is safe, unsafe or the resources are exhausted. We call this approach Refinement of Abstraction in Horn clauses using Finite Tree automata, RAHFT in short.

The approach just described lacks generalisation of spurious counterexamples during refinement. However, in our current approach, we generalise a spurious counterexample through the use of *interpolant automata*. Section 6.3 describes how to compute an *interpolant automaton* (taken from [147]) corresponding to an infeasible Horn clause derivation. We first construct an *interpolant automaton* viz. $\mathcal{A}^I_t$ corresponding to t. In Figure 6.4, this is shown by a blue line (in the middle) connecting the Abstraction and Refinement boxes. The refinement proceeds as in RAHFT with the only difference that $\mathcal{A}'_P$ now recognizes all traces in $\mathcal{L}(\mathcal{A}^M_P) \setminus \mathcal{L}(\mathcal{A}^I_t)$. We

call this approach Refinement of Abstraction in Horn clauses using Interpolant Tree automata, RAHIT in short.

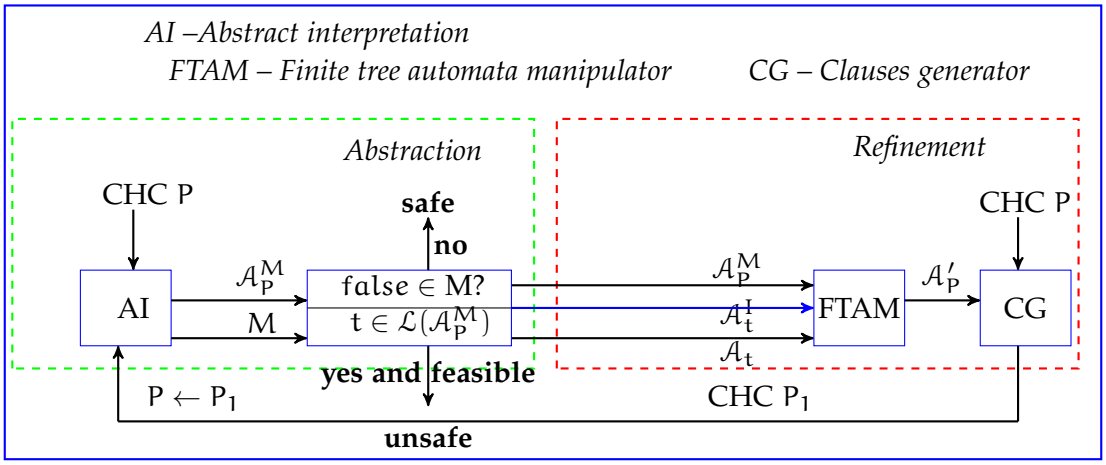

Figure 6.4: *Abstraction-refinement scheme in Horn clause verification [96]. M is an approximation produced as a result of abstract interpretation. $\mathcal{A}'_P$ recognizes all traces in $\mathcal{L}(\mathcal{A}_P^M) \setminus \mathcal{L}(\mathcal{A}_t)$.*

Next we briefly describe how to generate an FTA, $\mathcal{A}_P^M$, corresponding to a set of clauses P using the approximation produced by *abstract interpretation*. Finally we show some experimental results using our current approach on a set of Horn clause verification benchmarks.

OBTAINING AN FTA FROM A PROGRAM AND A MODEL. Let M be a set of constrained atoms of the form $p(X) \leftarrow \phi$ where p is a program predicate and ϕ is a constraint over X. Given such a set M, define γ_M to be the mapping from atoms to constraints such that $\gamma_M(p(X)) = \phi$ for each constrained fact $p(X) \leftarrow \phi$. M is a model of P (called a *syntactic solution* in [147]) if for each clause $p(X) \leftarrow \phi, p_1(X_1), \ldots, p_n(X_n)$ in P, $\phi \wedge \bigwedge_{i=1}^{n} \gamma_M(p_i(X_i)) \rightarrow \gamma_M(p(X))$.

Given such an M, we construct an FTA corresponding to P, which is the same as $\mathcal{A}_P$ (Definition 6.2) except that transitions corresponding to clauses whose bodies are not satisfiable in the model are omitted, since they cannot contribute to feasible derivations.

Definition 6.10 (FTA defined by a model.)**.** *Let P be a set of CHCs and M be a model defined by a set of constrained facts. Then the FTA $\mathcal{A}_P^M = (Q, Q_f, \Sigma, \Delta_M)$ where Q, Q_f and Σ are the same as for $\mathcal{A}_P$ (Definition 6.2) and Δ_M is the following set of transitions.*

$$\Delta_M = \{c(p_1, \ldots, p_n) \rightarrow p \mid \begin{array}{l} id_P(c) = p(X) \leftarrow \phi, p_1(X_1), \ldots, p_n(X_n), \\ SAT(\phi \wedge \bigwedge_{i=1}^{n} \gamma_M(p_i(X_i))) \end{array}$$

Lemma 6.1. *Let P be a set of clauses and M be a model of P then $\mathcal{L}(\mathcal{A}_P^M)$ includes all feasible trace-terms of P rooted at false.*

In our experiments, the abstraction interpretation was over the domain of convex polyhedra, yielding a set of constrained facts where each constraint is a conjunction of linear equalities and inequalities representing a convex polyhedron.

Example 6.5 (FTA produced as a result of *abstract interpretation*)**.** *For our example program in Figure 6.1, the convex polyhedral abstraction produces an over-approximation* M *which is represented as*

$$\mathrm{M} = \{fib(A, B) \leftarrow A >= 0, B >= 1, -A + B >= 0\}$$

Since there is no constrained fact for false in M*, this is a model for the example program. Our abstraction-refinement approach terminates at this point. However for the purpose of example, we show the FTA constructed for the example program using* M*. Since the bodies of each clause except the integrity constraint are satisfied under* M*, the FTA is same as the one depicted in Example 6.1 except the transition* $\mathtt{c}_3(\mathtt{fib}) \rightarrow \mathtt{false}$*, which is removed because of abstract interpretation.*

6.4.1 *Experiments*

For our experiment, we have collected a set of 68 programs from different sources.

1. A set of 30 programs from SV-COMP'15 repository[1] [13] (recursive category, Horn clauses derived from recursive C programs) and translated them to Horn clauses using inter-procedural encoding of SeaHorn [71, 70] producing (mostly) non-linear Horn clauses.

2. A set of 38 problems taken from the source repository[2], compiled by the authors of the tool Eldarica [80]. This set consists of problems, among others, from the NECLA static analysis suite, from the paper [87]. These tasks are also considered in [147] and are interpreted over *integer linear arithmetic*.

We made the following comparison between the tools.

1. We compare Rahit with Rahft, which compares the effect of removing a set of traces rather than a single trace.

2. We compare Rahit with the *trace-abstraction* tool [147] (*TAR* from now on). Rahit uses polyhedral approximation combined with trace abstraction refinement whereas *TAR* uses only trace abstraction refinement.

The results are summarized in Table 6.1.

1 http://sv-comp.sosy-lab.org/2015/benchmarks.php
2 https://github.com/sosy-lab/sv-benchmarks/tree/master/clauses/LIA/Eldarica

IMPLEMENTATION: Most of the tools in our tool-chain depicted in Figure 6.4 are implemented in Ciao Prolog [78] except the one for determinisation of FTA, which is implemented in Java following the algorithm described in [61]. Our tool-chain obtained by combining various tools using a *shell script* serves as a proof of concept which is not optimised at all. For handling constraints, we use the Parma polyhedra library [5] and the Yices SMT solver [44] over *linear real arithmetic*. The construction of tree interpolation uses constrained based algorithm presented in [138] for computing interpolant of two formulas.

DESCRIPTION: In Table 6.1, *Program* represents a verification task, *Time (secs) RAHFT* and *Time (secs) RAHIT* - respectively represent the time in seconds taken by the the tool RAHFT and RAHIT respectively for solving a given task. Similarly, the number of *abstraction-refinement* iteration needed in these cases to solve a task are represented by *#Itr. RAHFT* and *#Itr. RAHIT*. Similarly, *Time (secs) TAR* and *#Itr. TAR* represent the time taken and the number of iterations needed by the tool *TAR*. The experiments were run on a MAC computer running OS X on 2.3 GHz Intel core i7 processor and 8 GB memory.

DISCUSSION: The comparison between RAHFT and RAHIT would reflect purely the role of *interpolant tree automata* in Horn clause verification (Table 6.1) since the only difference between them is the refinement part using *(interpolant) tree automata*. The results show that RAHIT is more effective in practice than its counterpart RAHFT. This is justified by the number of tasks 61/68 solved by RAHIT using fewer iterations compared to RAHFT, which only solves 56/68 tasks. This is due to the generalisation of a spurious counterexample during refinement, which also captures other infeasible traces. Since these traces can be removed in the same iteration, it (possibly) reduces the number of refinements, however the solving time goes up because of the cost of computing an interpolant automaton. It is not always the case that RAHIT takes less iterations for a task (for example *Addition03 false-unreach*) than RAHFT. This is because the restructuring of the program obtained as a result of removing a set of traces may or may not favour polyhedral approximation. It is still not clear to us how to produce a right restructuring which favours polyhedral approximation. RAHIT times out on `cggmp2005_true-unreach` whereas RAHFT solves it in 5 iterations. We suspect that this is due to the cost of generating interpolant automata. We are not sure about the complexity of interpolant generation algorithm we used (the size of the formula generated was quite large with respect to the original program, magnitude not known) and there are several calls to the theorem prover to label each tree node with interpolants. So the bigger is the trace-tree, the longer it takes to compute the interpolant tree. On average, RAHIT needs 2.08 iterations and 11.40 seconds time to solve a task whereas RAHFT needs 2.32 iterations and 10.55 seconds.

The use of *interpolant tree automata* for trace generalisation and the tree automata based operations for trace-refinement are same in both RAHIT and *TAR*. Since *TAR* is not publicly available, we chose the same set of benchmarks used by *TAR* for the purpose of comparison and presented the results (the results corresponding to *TAR* are taken from [147]). The computer used in our experiments and in *TAR* [147] have similar characteristics. RAHIT solves more than half of the problems only with *abstract interpretation* over the domain of convex polyhedra without needing any refinement, which indicates its power. RAHIT solves 33/38

problems whereas *TAR* solves 28/38 problems. In average, RAHIT takes less time than *TAR*. In many cases *TAR* solves a task faster than RAHIT, however it spends much longer time in some tasks. Our current constraint solver is over *linear real arithmetic*, that is, a given program is safe/unsafe over reals. If we use it over *linear integer arithmetic* then the results may differ. We made some observation with the problems *boustrophedon.c*, *boustrophedon_expansed.c* and *cousot.correct* (which are supposed to be interpreted over integers). In them, if we replace strict inequalities ($>, <$) with non-strict inequalities ($\geqslant, \leqslant$) over integers (for example replace $X > Y$ with $X \geqslant Y + 1$), then we can solve them only with *abstract interpretation* without refinement which were not solved before the transformation using our solver. On the other hand, RAHIT times out for *mergesort.error* whereas *TAR* solves it in a single iteration. This indicates that the choice of a spurious counterexample and the quality of interpolant generated from it for generalisation have some effects on verification.

6.5 RELATED WORK

Horn Clauses, as an intermediate language, have become a popular formalism for verification [17, 59], attracting both the logic programming and software verification communities [16]. As a result of these, several verification techniques and tools have been developed for CHCs, among others, [70, 65, 94, 37, 96, 85, 80]. To the best of our knowledge, the use of automata based approach for *abstraction-refinement* of Horn clauses is relatively new [96, 147], though the original framework proposed for imperative programs goes back to [74, 75].

The work described in [96] uses FTA based approach for refining *abstract interpretation* over the domain of convex polyhedra [31], which is similar to *trace abstraction* [74, 77, 147] with the following differences. In [96], there is an interaction between state abstraction by *abstract interpretation* [32] and trace abstraction by FTA but there is no generalisation of spurious counterexamples. On one hand, [74, 77, 147] use *trace-abstraction* with the generalisation of spurious counterexamples using *interpolant automata* and may diverge from the solution due to the lack of right generalisation. On the other hand, *abstract interpretation* [32] is one of the most promising techniques for verification which is scalable but suffers from *false alarms*. When combined with refinement *false alarms* can be minimized. Our current work takes the best of both of these approaches.

6.6 CONCLUSION

This chapter brings together *abstract interpretation* over the domain of convex polyhedra and *interpolant tree automata* in an *abstraction-refinement* scheme for Horn clause verification and combines them in a new way. Experimental results on a set of software verification benchmarks using this scheme demonstrated their usefulness in practice; showing some slight improvements over the previous approaches. In the future, we plan to evaluate its effectiveness in a larger set of benchmarks, compare our approach with other similar approaches and improve the implementation aspects of our tool. Further study is needed to find a suitable combination of abstract interpretation and interpolation based techniques, based on a deeper

Program	Time (secs) RAHFT	#Itr. RAHFT	Time (secs) RAHIT	#Itr. RAHIT	Time (secs) TAR [147]	#Itr. TAR
addition	1	0	1	0	0.26	3
anubhav.correct	2	0	2	0	1.72	9
bfprt	1	0	1	0	0.43	6
binarysearch	2	0	2	0	0.36	5
blast.correct	5	1	11	1	8.93	65
boustrophedon.c	TO	-	TO	-	53.65	193
boustrophedon_expansed.c	TO	-	TO	-	69.06	340
buildheap	44	9	44	9	TO	-
copy1.error	11	0	11	0	12.79	19
countZero	1	0	1	0	TO	-
cousot.correct	TO	-	TO	-	TO	-
gopan.c	3	0	3	0	TO	-
halbwachs.c	TO	-	TO	-	TO	-
identity	1	0	1	0	7.67	34
inf1.error	4	1	9	1	0.51	6
inf6.correct	5	1	5	1	1.96	33
insdel.error	2	0	2	0	0.17	1
listcounter.correct	1	0	1	0	TO	
listcounter.error	9	1	9	1	0.21	1
listreversal.correct	4	0	4	0	35.79	149
listreversal.error	9	0	9	0	0.3	1
loop.error	3	0	3	0	3	3
loop1.error	8	0	8	0	10.87	19
mc91.pl	139	24	7	3	0.57	7
merge	2	0	2	0	0.86	10
mergesort.error	TO	-	TO	-	0.32	1
palindrome	2	0	2	0	0.61	6
parity	3	1	4	1	0.62	7
rate_limiter.c	3	0	3	0	49.96	130
remainder	1	0	1	0	1.5	17
running	3	1	8	2	0.4	5
scan.error	3	0	3	0	TO	-
string_concat.error	6	0	6	0	TO	-
string_concat1.error	TO	-	TO	-	TO	-
string_copy.error	3	0	3	0	TO	-
substring.error	5	0	5	0	0.55	1
substring1.error	15	0	15	0	2.84	5
triple	27	10	13	1	0.86	6
average (over 38)			**8.78**	**0.93**	**9.52**	**38.64**
solved/total			**33/38**	-	**28/38**	
Primes_true-unreach	16	4	4	1		
sum_10x0_false-unreach	5	2	12	2		
afterrec_false-unreach	2	1	3	1		
id_03_false-unreach	6	3	7	3		
cggmp2005_variant_true-unreach	2	1	3	1		
recHanoi01_true-unreach	8	3	10	3		
cggmp2005b_true-unreach	3	1	3	1		
gcd02_true-unreach	11	4	11	4		
diamond_false-unreach	3	1	3	1		
Addition03_false-unreach	6	2	13	5		
diamond_true-unreach-call1	2	1	3	1		
id_i5_05_false-unreach	19	8	12	5		
diamond_true-unreach-call2	6	1	5	1		
cggmp2005_true-unreach	10	5	TO	-		
gsv2008_true-unreach	3	1	3	1		
Fiboccio1_true-unreach	52	10	29	6		
id_b3_02_false-unreach	5	2	3	1		
Ackermann02_false-unreach	68	17	25	7		
mcmillan2006_true-unreach	2	1	3	1		
ddlm2013_true-unreach	TO	-	17	7		
sum_2x3_false-unreach	2	1	3	1		
fibo_5_true-unreach	TO	-	77	7		
Addition01_true-unreach	6	2	5	2		
Ackermann04_true-unreach	TO	-	59	8		
Addition02_false-unreach	4	2	5	2		
id_i10_010_false-unreach	TO	-	39	10		
gcd01_true-unreach	9	4	5	2		
id_010_false-unreach	TO	-	38	10		
gcnr2008_false-unreach	13	4	6	2		
Fiboccio4_false-unreach	TO	-	91	11		
average (over 68)	**10.55**	**2.32**	**11.40**	**2.08**		
solved/total	**56/68**		**61/68**			

Table 6.1: Experiments on software verification problems. In the table "TO" means time out which is set for 300 seconds, "-" indicates the insignificance of the result.

understanding of the interaction among interpolation, trace elimination and abstract interpretation.

7

DECOMPOSITION BY TREE DIMENSION IN HORN CLAUSE VERIFICATION

With John P. Gallagher and Pierre Ganty

Abstract
In this chapter we investigate the use of the concept of *tree dimension* in Horn clause analysis and verification. The dimension of a tree is a measure of its non-linearity – for example a list of any length has dimension zero while a complete binary tree has dimension equal to its height. We apply this concept to trees corresponding to Horn clause derivations. A given set of Horn clauses P can be transformed into a new set of clauses $P^{\leqslant k}$, whose derivation trees are the subset of P's derivation trees with dimension at most k. Similarly, a set of clauses $P^{>k}$ can be obtained from P whose derivation trees have dimension at least $k+1$. In order to prove some property of all derivations of P, we systematically apply these transformations, for various values of k, to decompose the proof into separate proofs for $P^{\leqslant k}$ and $P^{>k}$ (which could be executed in parallel). We show some preliminary results indicating that decomposition by tree dimension is a potentially useful proof technique. We also investigate the use of existing automatic proof tools to prove some interesting properties about dimension(s) of feasible derivation trees of a given program.

Keywords: Tree dimension, proof decomposition, program transformation, Horn clauses.

7.1 INTRODUCTION

In this chapter, we study the role of *tree dimension* in Horn clause analysis and verification. The dimension of a tree is a measure of its non-linearity – for example a list of any length has dimension zero while a complete binary tree has dimension equal to its height. We apply this concept to trees corresponding to Horn clause derivations. A given set of Horn clauses P can be transformed into a new set of clauses $P^{\leqslant k}$ (whose derivation trees are the subset of P's derivation trees with dimension at most k) and $P^{>k}$ (whose derivation trees have dimension at least $k+1$). Each such set of clauses represents an under-approximation of the original set of clauses and the proof for the original clauses can be constructed from their individual proofs. In order to prove some property of all derivations of P, we systematically apply these transformations, for various values of k, to decompose the proof into separate proofs for $P^{\leqslant k}$ and $P^{>k}$ (which could be executed in parallel). This decomposition is motivated by the work by Esparza et al. [46]. They claim that analysis of programs by reasoning about dimension of their derivation trees provides certain advantage over reasoning about height of their derivation trees.

```
c1. fib(A, A):- A>=0,  A=<1.
c2. fib(A, B) :- A > 1, A2 = A - 2, fib(A2, B2),
           A1 = A - 1, fib(A1, B1), B = B1 + B2.
c3. false:- A>5, fib(A,B), B<A.
```

Figure 7.1: Example CHCs Fib: it defines a Fibonacci function.

We prove each such set of clauses using abstract interpretation [32] over the domain of convex polyhedra [31] as described in [94]. Finally, the preliminary results in a set of Horn clause verification benchmarks show that this is a useful program transformation. This decomposition can also be viewed as refinement where one eliminates possibly infinite sets of program traces. As a result of this, the proof for the remaining part becomes simpler. To motivate readers, we present an example set of constrained Horn clauses (CHCs) P in Figure 7.1 which defines the Fibonacci function. This is an interesting problem whose dimension depends on the input number and its computations are trees rather than linear sequences. The main contributions of this chapter are the following.

1. We describe how to generate at-most k-dimension program and at-least k-dimension program from a given program using the notion of tree dimension (Section 7.2);
2. We give a verification algorithm for Horn clauses program based on its proof decomposition (Section 7.3);
3. We give an alternative way of generating the at-least k-dimension program using the theory of finite tree automata (Section 7.4);
4. We demonstrate the feasibility of our approach in practice applying it to non-linear Horn clause verification problems (Section 7.7);
5. We instrument a program with its dimension and use existing automatic verification tools to prove some interesting properties about its dimension (Section 7.5).

7.2 PRELIMINARIES

A constrained Horn clause is a first order formula of the form $p(X) \leftarrow \mathcal{C}, p_1(X_1), \ldots, p_k(X_k)$ ($k \geqslant 0$) (using Constraint Logic Programming (CLP) syntax), where $\mathcal{C}$ is a conjunction of constraints with respect to some background theory, X_i, X are (possibly empty) vectors of distinct variables, $p_1, \ldots, p_k, p$ are predicate symbols, $p(X)$ is the head of the clause and $\mathcal{C}, p_1(X_1), \ldots, p_k(X_k)$ is the body. A clause is called non-linear if it contains more than one atom in the body ($k > 1$), otherwise it is called linear. A set of Horn clauses is sometimes called a program.

A labeled tree $c(t_1, \ldots, t_k)$ is a tree with its nodes labeled, where c is a node label and $t_1, \ldots, t_k$ are labeled trees rooted at the children of the node and leaf nodes are denoted by c.

Definition 7.1 (Tree dimension (adapted from [45]))**.** *Given a labeled tree* $t = c(t_1, \ldots, t_k)$*, the tree dimension of* t *represented as* $dim(t)$ *is defined as follows:*

$$dim(t) = \begin{cases} 0 & \text{if } k = 0 \\ \max_{i \in [1..k]} dim(t_i) & \text{if there is a unique maximum} \\ \max_{i \in [1..k]} dim(t_i) + 1 & \text{otherwise} \end{cases}$$

Figure 7.2 (a) shows a derivation tree t for Fibonacci number 3 and Figure 7.2 (b) shows its tree dimension. It can be seen that $dim(t) = 1$. This number is a measure of its non-linearity, the smaller the number the closer the tree is to a list. Since it is not a perfect binary tree, the height of t (3) is greater than its dimension.

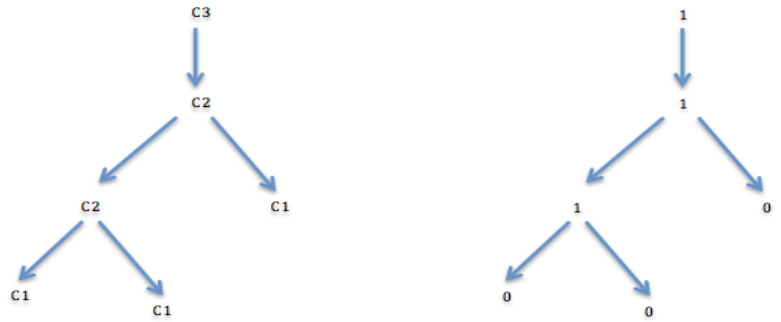

Figure 7.2: (a) derivation tree of Fibonacci 3 and (b) its tree dimension.

Given a set of CHCs P and $k \in \mathbb{N}$, we split each predicate H occurring in P into the predicates $H^{\leqslant d}$ and $H^{=d}$ where $d \in \{0, 1, \ldots, k\}$. Here $H^{\leqslant d}$ and $H^{=d}$ generate trees of dimension at most d and exactly d respectively.

Definition 7.2 (At-most-k-dimension program $P^{\leqslant k}$)**.** *It consists of the following clauses (adapted from [121]):*

1. *Linear clauses:*

   If $H \leftarrow \mathcal{C} \in P$ *, then* $H^{=0} \leftarrow \mathcal{C} \in P^{\leqslant k}$.

   If $H \leftarrow \mathcal{C}, B_1 \in P$ *then* $H^{=d} \leftarrow \mathcal{C}, B_1^{=d} \in P^{\leqslant k}$ *for* $0 \leqslant d \leqslant k$.

2. *Non-linear clauses:*

   If $H \leftarrow \mathcal{C}, B_1, B_2, \ldots, B_r \in P$ *with* $r > 1$*:*

   a) *For* $1 \leqslant d \leqslant k$*, and* $1 \leqslant j \leqslant r$*:*

      Set $Z_j = B_j^{=d}$ *and* $Z_i = B_i^{\leqslant d-1}$ *for* $1 \leqslant i \leqslant r \wedge i \neq j$*. Then:* $H^{=d} \leftarrow \mathcal{C}, Z_1, \ldots, Z_r \in P^{\leqslant k}$.

   b) *For* $1 \leqslant d \leqslant k$*, and* $J \subseteq \{1, \ldots, r\}$ *with* $|J| = 2$*:*

      Set $Z_i = B_i^{=d-1}$ *if* $i \in J$ *and* $Z_i = B_i^{\leqslant d-1}$ *if* $i \in \{1, \ldots, r\} \setminus J$*. If all* Z_i *are defined, i.e.,* $d \geqslant 2$ *if* $r > 2$*, then:* $H^{=d} \leftarrow \mathcal{C}, Z_1, \ldots, Z_r \in P^{\leqslant k}$.

```
%linear clauses
1. fib(0)(A,A) :- A>=0, A=<1.
2. false(0) :- A>5, B<A, fib(0)(A,B).
%epsilon-clauses
3. false[0] :- false(0).
4. fib[0](A,B) :- fib(0)(A,B).
```

Figure 7.3: $\mathrm{Fib}^{\leqslant 0}$: at-most 0-dimension program of Fib.

3. *ϵ-clauses:*

   $H^{\leqslant d} \leftarrow H^{=e} \in P^{\leqslant k}$ *for* $0 \leqslant d \leqslant k$ *, and every* $0 \leqslant e \leqslant d$.

Let nl and l be the number of non-linear and linear clauses in P, r be the max number of non-constraint atoms in the body of clauses in P and a be the number of non-constraint atoms of P. Then the size of $P^{\leqslant k}$ is given as follows. There are $l * (k+1)$ linear clauses, $nl * k * r$ number of non-linear clauses of the first kind (Point 2(a) of Definition 7.2), $nl * k * r * (r-1)/2$ number of non-linear clauses of the second kind (Point 2(b) of Definition 7.2) and $a * (k+1) * (k+2)/2$ ϵ-clauses.

The at-most 0-dimension program of Fib in Figure 7.1 is depicted in Figure 7.3 (where the numbers on the first column are not clause identifiers and are there for future reference). In textual form we represent a predicate $p^{\leqslant k}$ by `p[k]` and a predicate $p^{=k}$ by `p(k)`. Since some programs have derivation trees of unbounded dimension, trying to verify a property for its increasing dimension separately is not a practical strategy. To deal with this, we need some construction which characterises derivation trees of at-least k-dimension. Next we define this construction (*at-least k-dimension program*). For this, we split each predicate H occurring in P into the predicates $H^{>d}$ and $H^{\geqslant 0}$ where $d \in \{0, 1 \ldots, k\}$. Here $H^{>d}$ generates trees of dimension at-least $d+1$ and $H^{\geqslant 0}$ generates trees of any dimension.

Definition 7.3 (At-least k+1-dimension program $P^{>k}$)**.** *In addition to the linear, non-linear and ϵ-clauses from Definition 7.2 (with each predicate $H^{\leqslant k+1}$ and $H^{=k+1}$ from $P^{\leqslant k+1}$ renamed to $H^{\geqslant 0}$ and $H^{>k}$ respectively), the at-least k+1-dimension program $P^{>k}$ consists of the following clauses:*

1. *Link-clauses:*

   For each $H \leftarrow B \in P$ *there is a clause* $H^{\geqslant 0} \leftarrow H \in P^{>k}$.

2. *Original clauses:*

   All clauses in P *are also in* $P^{>k}$.

7.3 PROCEDURE FOR VERIFICATION

Given a set of CHCs P (including clauses with *false* head, also known as *integrity constraints*), the CHC verification problem is to check whether there exists a model of P. This is equivalent to checking whether there is any feasible derivation tree for *false*; if there is such a derivation then there is no model. We say P is safe if it has a model and unsafe if it has no model.

Algorithm 7.1: Verification algorithm for Horn clauses

```
1  Procedure VERIFY (P)
   Input: Set of CHCs P
   Output: safe, unsafe, unknown
2  initialization: k ← 0
3  while true do
4      generate P^≤k
5      r_1 ← SAFE(P^≤k)
6      if r_1 ≠ safe then
7          return r_1
8      generate P^>k
9      r'_1 ← SAFE( P^>k)
10     if r'_1 ≠ unknown then
11         return r'_1
12     k ← k + 1
```

The procedure VERIFY(P) is described in algorithm 1. VERIFY makes use of the procedure SAFE(P) in Algorithm 1, which is an oracle that returns *safe*, *unsafe* or *unknown*. The oracle is sound: if SAFE(P) returns *safe* (*unsafe*) then P is safe (unsafe). SAFE could be any existing automatic Horn clause solver [64, 95, 94, 79, 37]. When it cannot verify a program within a given time limit, the *unknown* answer is emitted. A given set of Horn clauses P can be transformed into a new set of clauses $P^{\leqslant k}$ and $P^{>k}$. In order to prove some property of all derivations of P, we systematically apply these transformations, for various values of k, to decompose the proof into separate proofs for $P^{\leqslant k}$ *(line 4)* and $P^{>k}$ *(line 9)*; which can be done in parallel. If both are safe then P is *safe*. If one of them is unsafe then P is *unsafe*. If an oracle cannot prove whether $P^{\leqslant k}$ is *safe/unsafe* then we return an *unknown* answer (we assume that the oracle would also return *unknown* for larger values of k). But if it cannot prove whether $P^{>k}$ is *safe/unsafe* then we try the *while loop* in the algorithm 1 with $k = k + 1$.

One possible optimisation that we can make in Algorithm 1 is to consider $P^{>k}$ instead of P in the next iteration of the *while loop* if we reach *line 14*. This is because at this stage we have already proven the safety of $P^{\leqslant k}$.

The soundness of Algorithm 1 is captured by the following lemma and proposition.

Lemma 7.1 (Decomposition by dimension)**.** *For all* k*, program* P *is safe if and only if both* $P^{\leqslant k}$ *and* $P^{>k}$ *are safe.*

Proposition 7.1 (Soundness)**.** *If Algorithm 1 returns* safe *then the input program is safe. If it returns* unsafe *then the program is unsafe.*

7.4 DIMENSION DECOMPOSITION USING FINITE TREE AUTOMATA

In this section, we show an alternative method for constructing an at-least k-dimension program, using operations on finite tree automata (FTAs). We first describe the connection between Horn clauses and FTAs and show how to construct an FTA from a set of Horn clauses.

7.4.1 *Trace automata for CHCs*

We add identifiers to clauses, whose purpose is to act as constructors of trace trees representing derivations. The identifiers are chosen from a set Σ of ranked function symbols. If P is a set of CHCs, let $\mathsf{id}_P : P \to \Sigma$ be an assignment of function symbols to clauses, such that for every clause $\mathtt{cl} \in P$, the arity of $\mathsf{id}_P(\mathtt{cl})$ equals the number of atoms in the body of $\mathtt{cl}$. We allow the same symbol to be assigned by id_P to more than one clause. We can also identify the predicates whose derivations are of interest (the *accepting* predicates in Definition 7.4).

Definition 7.4 (Trace FTA for a set of CHCs)**.** *Let P be a set of CHCs, Σ be a set of ranked function symbols and $\mathsf{id}_P : P \to \Sigma$ be a mapping from clauses to function symbols of appropriate arity. Let F be a set of predicates from P called the* accepting *predicates. Define the trace FTA for P as $\mathcal{A}^F_P = (Q, F, \Sigma, \Delta)$ where*

- *Q is the set of predicate symbols of P;*
- *$F \subseteq Q$ is the set of accepting predicate symbols;*
- *Σ is a set of function symbols;*
- *$\Delta = \{c(p_1, \ldots, p_k) \to p \mid \mathtt{cl} \in P,\ \mathtt{cl} = p(X) \leftarrow \mathcal{C}, p_1(X_1), \ldots, p_k(X_k),\ c = \mathsf{id}_P(\mathtt{cl})\}$.*

If F is the set of all predicate symbols occurring in the clauses we omit the superscript F from $\mathcal{A}^F_P$.

The set of trees accepted by $\mathcal{A}^F_P$ is written $\mathcal{L}(\mathcal{A}^F_P)$. Elements of $\mathcal{L}(\mathcal{A}^F_P)$ are called the trace trees for P. $\mathcal{L}(\mathcal{A}^F_P)$ is isomorphic to the set of (successful and unsuccessful) derivation trees (for atomic formulas with accepting predicates) constructible from P and from now on we identify trace trees with derivations. We do not define derivation trees formally here, but refer to the notion of an AND-tree in the literature [142, 56].

Example 7.1. *Let P be the set of CHCs in Figure 7.1 and let $F = \{\mathtt{fib}, \mathtt{false}\}$. Let id_P map the clauses to c_1, c_2, c_3 respectively. Then $\mathcal{A}^F_P = (Q, F, \Sigma, \Delta)$ where:*

$$
\begin{array}{rclcl}
Q & = & \{\mathtt{fib}, \mathtt{false}\} & \quad \Delta & = \{c_1 \to \mathtt{fib}, \\
\Sigma & = & \{c_1, c_2, c_3\} & & \phantom{=\{}c_2(\mathtt{fib}, \mathtt{fib}) \to \mathtt{fib}, \\
 & & & & \phantom{=\{}c_3(\mathtt{fib}) \to \mathtt{false}\}
\end{array}
$$

Figure 7.2(a) shows a trace tree recognised by this FTA. The tree can also be written as the trace-term $c_3(c_2(c_2(c_1, c_1), c_1))$.

If a mapping $\mathsf{id}_P : P \to \Sigma$ assigns a unique identifier to each clause, that is, id_P is injective, then there is an inverse mapping $\mathsf{id}^{-1} : \mathsf{range}(\mathsf{id}_P) \to P$.

Definition 7.5 ($\mathsf{chc}_{\mathsf{id}}(\mathcal{A})$)**.** *Given an FTA* $\mathcal{A} = (Q, F, \Sigma, \Delta)$ *and an injective mapping* id *such that* $\Sigma \subseteq \mathsf{range}(\mathsf{id})$*, we can construct a set of CHCs from* $\mathcal{A}$*, called* $\mathsf{chc}_{\mathsf{id}}(\mathcal{A})$*, defined as follows:*

$$\mathsf{chc}_{\mathsf{id}}(\mathcal{A}) = \{q(X) \leftarrow \mathcal{C}, q_1(X_1), \ldots, q_n(X_n) \mid \begin{array}{l} c(q_1, \ldots, q_n) \to q \in \Delta, \\ \mathsf{id}^{-1}(c) = q(X) \leftarrow \mathcal{C}, q_1(X_1), \ldots, q_n(X_n)\} \end{array}$$

The set of accepting predicates of $\mathsf{chc}_{\mathsf{id}}(\mathcal{A})$ *is defined to be* F*.*

In the definitions we reuse the states in the FTA as predicate symbols in the constructed clauses. In practice we use some injective renaming function from states to predicates in the constructed program. Further discussion of the mappings between CHCs and FTAs can be found in [95]. By construction, the derivations of $\mathsf{chc}_{\mathsf{id}}(\mathcal{A})$ (for the accepting predicates) correspond to the elements of $\mathcal{L}(\mathcal{A})$.

7.4.2 *Construction of the at-least k-dimensional program using FTA operations*

In the construction of the at-least k-dimension program $P^{>k}$ in Definition 7.3, the original program clauses from P are included in the generated clauses. The presence of the original clauses suggests that the "decomposed" verification problem for $P^{>k}$ is as hard as the original problem for P, since it contains the clauses of P as well as others, and so this form might not lend itself to verification.

Thus in the following construction we build $P^{>k}$ based on FTA language difference, and the original clauses are not copied to the at-least k-dimension program. We first define a general FTA-difference for CHCs.

Definition 7.6 (FTA-difference for CHCs)**.** *Let* P *and* Q *be sets of CHCs,* F_1 *and* F_2 *their respective accepting predicates and* $\mathsf{id}_P : P \to \Sigma$ *and* $\mathsf{id}_Q : Q \to \Sigma$ *their respective identifier assignments, where* id_P *is injective. Let* $\mathcal{A}_P^{F_1}$ *and* $\mathcal{A}_Q^{F_2}$ *be the trace FTAs constructed from* P, Q *respectively. Then the FTA-difference of* P *and* Q *(with their respective accepting predicates) written* $P^{F_1} - Q^{F_2}$*, is given as* $\mathsf{chc}_{\mathsf{id}_P}(\mathcal{A}_P^{F_1} \setminus \mathcal{A}_Q^{F_2})$ *where* $\setminus$ *is the difference of FTAs [28]. The set of accepting predicates is the set of accepting states for the difference FTA.*

The set of derivations for $P^{F_1} - Q^{F_2}$ contains, by construction, those derivations of P^{F_1} that are not derivations of Q^{F_2}. We now apply these notions to the verification procedure based on decomposition. We are given a set of CHCs P, with accepting predicates $F = \{\mathit{false}\}$. In the program $P^{\leqslant k}$, the set of accepting predicates is $F^k = \{\mathit{false}^{\leqslant k}\}$. Note that we can ignore the derivations for the other predicates of the form $\mathit{false}^{\leqslant j}$ or $\mathit{false}^{=j}$ since $\mathit{false}^{\leqslant k}$ by construction accumulates their derivations, for all $j \leqslant k$.

7.4.2.1 *Assignment of identifiers in the at-most-k-dimension program*

Given a program P and the at-most-k-dimension program $P^{\leqslant k}$, we intend to construct the difference $P^{\{\mathit{false}\}} - P^{\leqslant k\{\mathit{false}^{\leqslant k}\}}$ using Definition 7.6. In order to do so, we first need to con-

struct the identifier assignment $\mathsf{id}_{P^{\leqslant k}}$ so as to preserve trace trees from P. This requires the modification of $P^{\leqslant k}$ to eliminate the ϵ-clauses, as follows.

Definition 7.7 (Unfolding of ϵ-clauses in $P^{\leqslant k}$)**.** *Let $P^{\leqslant k}$ be the at-most-k-dimension program obtained from P using Definition 7.2. Replace each ϵ-clause of form $H^{\leqslant d} \leftarrow H^{=e}$ by the set of clauses $H^{\leqslant d} \leftarrow B$, where $H^{=e} \leftarrow B$ is either a linear or non-linear clause in $P^{\leqslant k}$.*

The elimination of ϵ-clauses is an instance of the well-known unfolding transformation which preserves the derivability of atomic formulas. In other words an atom A is derivable from a program P if and only if it is derivable after applying the unfolding transformation [131].

In the following definition, the clause identifiers are chosen for clauses in $P^{\leqslant k}$ Informally, every clause of $P^{\leqslant k}$ inherits the clause identifier for the clause in P from which it originates. More precisely we define the clause identifiers for $P^{\leqslant k}$ as follows.

Definition 7.8 (Assignment of clause identifiers in $P^{\leqslant k}$)**.** *Let $P^{\leqslant k}$ be the at-most-k-dimension program obtained from P using Definition 7.2, with ϵ-clauses eliminated according to Definition 7.7. Each clause of $P^{\leqslant k}$ is a linear, non-linear or an ϵ-unfolded-clause. The clause identifiers are assigned in two steps as follows.*

1. *Assign to each linear or non-linear clause the clause identifier from the clause in P from which it is derived in Definition 7.2.*
2. *Assign to each unfolded ϵ-clause the clause identifier for the linear or non-linear clause used to unfold it using Definition 7.7.*

We are now in a position to compare the sets of trace trees for P and $P^{\leqslant k}$ using their respective FTAs.

Lemma 7.2. *Let P be a set of CHCs and let $\mathsf{id}_P : P \to \Sigma$ be an injective function assigning clause identifiers to P. Let $F_1 = \{false\}$. Let $k \geqslant 0$ and let $P^{\leqslant k}$ be the at-most-k-dimension program obtained from P using Definition 7.2 with ϵ-clauses unfolded using Definition 7.7 and let $F_2 = \{false^{\leqslant k}\}$. Then $\mathcal{L}(\mathcal{A}^{F_2}_{P^{\leqslant k}}) = \{t \mid t \in \mathcal{L}(A^{F_1}_P), dim(t) \leqslant k\}$.*

The proof is by induction on derivations in $P^{\leqslant k}$ and uses the correspondence of the clause identifiers as set up in Definition 7.8.

Theorem 7.1. *Let P be a set of CHCs and let $\mathsf{id}_P : P \to \Sigma$ be an injective function assigning clause identifiers to P. Let $k \geqslant 0$ and let $P^{\leqslant k}$ be the at-most-k-dimension program obtained from P using Definition 7.2 with ϵ-clauses unfolded using Definition 7.7. Then false is derivable from $P - P^{\leqslant k}$ if and only if $false^{>k}$ is derivable from $P^{>k}$.*

Thus we have shown a different method of constructing the at-least k-dimension program $P^{>k}$, namely by taking the difference of P with $P^{\leqslant k}$, which contains only derivations (for its accepting predicates) that have dimension greater than k.

Details on difference construction can be found in [95]. We construct the difference of two FTAs by (1) standardising apart the predicate names; (2) forming the union of the two FTAs;

```
c1. fib(0)(A,A) :- A>=0, A=<1.
c3. false(0) :- A>5, B<A, fib(0)(A,B).
c3. false[0] :- A>5, B<A, fib(0)(A,B).
c1. fib[0](A,B) :-A>=0, A=<1.
```

Figure 7.4: $\text{Fib}^{\leqslant 0}$ after unfolding ϵ-clauses and assigning clause identifiers.

(3) determinising the union; (4) removing from the determinised FTA all states (and transitions that contain them) that contain an accepting state of the second FTA. Note that the set of states of the determinised FTA is a subset of the powerset of the original states. Note that determinisation of FTAs is often considered prohibitively complex even for small FTAs. We use a recent optimised FTA determinisation algorithm [60], returning a compact form of the determinised called product form, which can be used directly in constructing the resulting clauses.

Example 7.2. *We illustrate this through an example using* $\text{Fib}^{\leqslant 0}$ *(Figure 7.3). The clauses 1 and 2 in* $\text{Fib}^{\leqslant 0}$*, will have* c_1 *and* c_3 *as identifiers since they were derived respectively from the clauses* c_1 *and* c_3 *in Fib (Figure 7.1). By unfolding* ϵ*-clauses (clauses 3 and 4) using respectively clauses 2 and 1 in Figure 7.3, we obtain* `false[0] :- A>5, B<A, fib(0)(A,B) and fib[0](A,B) :-A>=0, A=<1`*. They will have identifiers* c_3 *and* c_1 *respectively. Therefore, the clauses in* $\text{Fib}^{\leqslant 0}$ *will have the identifiers assigned as shown in Figure 7.4.*

After assigning identifiers to each of the clauses in $\text{Fib}^{\leqslant 0}$, we can construct an FTA corresponding to it using Definition 7.4, and obtain the FTA shown in Figure 7.5: as before we represent a predicate $p^{\leqslant k}$ by `p[k]` and a predicate $p^{=k}$ by `p(k)`.

$$
\begin{aligned}
&Q = \{\texttt{fib(0)},\ \texttt{false(0)},\ \texttt{false[0]},\ \texttt{fib[0]}\} \qquad \Delta = \{c_1 \rightarrow\ \texttt{fib(0)},\\
&F = \{\texttt{false[0]}\} \qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ c_3(\texttt{fib(0)}) \rightarrow \texttt{false(0)},\\
&\Sigma = \{c_1, c_3\} \qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ c_3(\texttt{fib(0)}) \rightarrow \texttt{false[0]},\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad\ \ c_1 \rightarrow \texttt{fib[0]}\}
\end{aligned}
$$

Figure 7.5: FTA (Q, F, Σ, Δ) corresponding to $\text{Fib}^{\leqslant 0}$.

The difference FTA between $\mathcal{A}^{\{\mathit{false}\}}_{\texttt{Fib}}$ and $\mathcal{A}^{\{\mathit{false}^{\leqslant 0}\}}_{\texttt{Fib}^{\leqslant 0}}$ accepts trees rooted at *false* which have dimension greater than 0. The determinised FTA (DFTA) constructed as explained above is shown in the Figure 7.6. DFTA states are sets of predicates, and we represent a set using square brackets instead of curly brackets in the code, e.g. `[fib(0), fib[0], fib]`. Furthermore the product form referred to above contains set of DFTA states, such as `[[fib(0), fib[0], fib], [fib]]`.

We can generate a new program from this DFTA together with the original program Fib following the approach taken in [95] obtaining the program in Figure 7.7. It should be noted that the derivation trees rooted at *false* have dimension at-least 1. Now verification of the

```
c1 -> [fib(0), fib[0], fib].
c2([[fib(0), fib[0], fib], [fib]],
                      [[fib(0), fib[0], fib], [fib]]) -> [fib].
c3([[fib]]) -> [false].
```

Figure 7.6: Transitions of the determinised FTA.

original program Fib is decomposed into verifying the program in Figure 7.3 (where `false[0]` is replaced by `false` and the program in Figure 7.7.

```
fib_0(A,A) :- A>=0, A=<1.
fib(A,B) :-  A>1,C=A-2,D=A-1,B=E+F,fib_1(C,F), fib_1(D,E).
false :- A>5, B<A, fib(A,B).
fib_1(A,B) :- fib_0(A,B).
fib_1(A,B) :-  fib(A,B).
```

Figure 7.7: At-least 1-dimension program of Fib produced using the difference of FTAs

7.5 PROGRAM INSTRUMENTATION WITH DIMENSION

The dimension of successful derivations in a set of CHCs is not always obvious from the text of the clauses. In some cases a bound on the dimension is clear from the form of the clauses; for instance all derivations using a set of linear clauses clearly have dimension zero. But consider the well known 91-function of McCarthy[1], represented in Figure 7.8 using Horn clauses.

Although it is possible to construct derivation trees of arbitrary dimension using the clauses in Figure 7.8, the dependencies between the two recursive calls to `mc91` imply that no *successful* derivation has dimension greater than 2. We now show how to establish this using a transformation to instrument the clauses with dimension information, and then use automatic verification tools to establish properties of the dimension.

Definition 7.9 (Dimension-instrumented clauses)**.** *Let* P *be a set of CHCs. Define the set* P_{dim} *of CHC as follows.*

- *For each predicate* p *of arity* m *define a predicate* p' *of arity* $m+1$*.*
- *For each clause in* P *of the form*

$$p(X) \leftarrow \mathcal{C}, p_1(X_1), \ldots, p_n(X_n)$$

1 http://en.wikipedia.org/wiki/McCarthy_91_function

```
mc91(N,X) :-  N > 100,  X = N-10.
mc91(N,X) :-  N =< 100,  Y = N+11,
              mc91(Y,Y2), mc91(Y2,X).
```

Figure 7.8: McCarthy's 91-function defined as Horn clauses

construct a clause

$$p'(X,K) \leftarrow \mathcal{C}, p_1'(X_1,K_1),\ldots,p_n'(X_n,K_n), \mathtt{dim}_n(K_1,\ldots,K_n,K)$$

in $P_{\mathtt{dim}}$*, where* $K_1,\ldots,K_n,K$ *are variables added as the final argument for their respective predicates, and* $\mathtt{dim}_n(K_1,\ldots,K_n,K)$ *is defined according to the rules in Definition 7.1 for determining the dimension of a tree.*

Example 7.3. *The dimension-instrumented version of the McCarthy 91-function contains the following clauses.*

```
mc91(N,X,K) :-  N >100,  X=N-10, dim0(K).
mc91(N,X,K) :-  N =<100,  Y=N+11,
     mc91(Y,Y2,K1), mc91(Y2,X,K2), dim2(K1,K2,K).
dim0(K):-K=0.
dim2(K1, K2, K3):-K1>=K2+1, K3=K1.
dim2(K1, K2, K3):-K2>=K1+1, K3=K2.
dim2(K1, K2, K3):- K1=K2, K3 = K1+1.
```

Using the instrumented program we can try to get information about the dimension, such as upper or lower bounds or other relationships between the dimension and other predicate arguments. It follows from the undecidability result of Gruska [66] on context-free grammars, that the problem of determining whether the dimension of set of CHC is bounded by a constant is, in general, undecidable.

Example 7.4. *To establish that the upper bound of successful derivations is 2, for facts* `mc91(X,Y)`*, we add the following integrity constraint to the dimension-instrumented clauses.*

```
false :- K > 2, mc91(X,Y,K).
```

The clauses together with the integrity constraint are given to an automatic solver for Horn clauses [64, 95], which is able to prove the safety of the clauses and thus establish the upper bound of 2.

In the next example, we show that the dimension can depend on the values of other predicate arguments.

Example 7.5. *The dimension-instrumented version of the* Fib *clauses is shown in Figure 7.9. The property to be proved is that the dimension of* Fib *is lesser or equal to the half of its input value, expressed by the integrity constraint* `false:- fib(A,B, K), 2*K -1>=A`*. Again, this property is established by applying a Horn clause solver to prove the safety of the clauses together with the integrity constraint.*

Example 7.6. *We present the well known counting change example taken from [2, Chapter 1]. The Figure 7.10 shows its CLP encoding and the Figure 7.11 shows the dimension-instrumented version in CLP. The property of interest is to relate the number of different coins (counts) with the program dimension. We can establish that the dimension is at most the number of different coins as expressed by the integrity constraint* `false :- B>=1, K > B, cc(A, B, C, K)`.

```
fib(A, A, K):-  A>=0,  A=<1, dim0(K).
fib(A, B, K) :- A>1, A2=A-2, fib(A2, B2, K1),
          A1= A-1, fib(A1, B1, K2), B=B1+B2, dim2(K1, K2, K).
dim0(K):-K=0.
dim2(K1, K2, K3):-K1>=K2+1, K3=K1.
dim2(K1, K2, K3):-K2>=K1+1, K3=K2.
dim2(K1, K2, K3):- K1=K2, K3=K1+1.
```

Figure 7.9: Fib program instrumented with its dimension

```
% base case: that is a hit
cc(0, Y, 1) :- Y>0.
% base case: that is a miss
cc(X, _, 0) :- X<0.
cc(_, Y, 0) :- Y=<0.
%inductive case
cc(X, Y, Z) :- X>0, kinds_of_coins(Y,A),
                    X1=X-A, cc(X1, Y, Z1),
                    Y1=Y-1, cc(X, Y1, Z2), Z=Z1+Z2.
kinds_of_coins(A,B) :- A >=1, B>=1.
```

Figure 7.10: Counting change example encoded as CLP clauses

```
cc(0, Y, 1,K) :- Y>0, dim0(K).
cc(X, _, 0,K) :- X<0, dim0(K).
cc(_, Y, 0,K) :- Y=<0, dim0(K).
cc(X, Y, Z,K) :-
        X>0, kinds_of_coins(Y,A, K0), X1=X-A,
        cc(X1, Y, Z1,K1), Y1=Y-1, cc(X, Y1, Z2,K2),
        Z = Z1+Z2, dim3(K0, K1,K2,K).
kinds_of_coins(A,B, K) :- A >= 1, B >= 1, dim0(K).
dim3(K0, K1,K2,K):-
        dim2(K0, K1, K3), dim2(K3,K2, K).
%predicates dim0(K) and dim2(K1, K2, K) are defined as above
```

Figure 7.11: Counting change example instrumented with its dimension

In general, verifying whether a program has a certain dimension is as challenging as proving any other properties of the program. But in some cases the knowledge of program dimension is useful for proving other program properties. For instance, using the knowledge that the McCarthy 91-function has dimension at most 2 would allow us to restrict the proof of any program property relating to successful derivations to the program $P^{\leqslant 2}$ where P is the set of clauses for the McCarthy 91-function.

Program	Result	Time(s)	dim(k)	Program	Result	Time(s)	dim(k)
addition	safe	4	0	fib	safe	4	0
bfprt	safe	4	0	mc91	safe	4	0
binarysearch	safe	4	0	revlen	safe	4	0
countzero	safe	3	0	running	unsafe	6	1
floodfill	safe	3	0	triple	unsafe	-	-
identity	safe	4	0	buildheap	unsafe	-	-
merge	safe	5	0	parity	unsafe	4	0
palindrome	safe	3	0	remainder	unsafe	4	0
average		4					

Table 7.1: Experimental results on non-linear CHC verification problems

7.6 RELATED WORK

The notion of dimension of a tree has a long history in science (starting with Geology) which has been detailed by Esparza *et al.* [48]. However, the use of dimension for program verification is more recent. Ganty and Iosif used it [63] for computing summaries of programs with procedures whose variables (global, local and parameters) take their value from the set of integers. Roughly speaking, the method they define first computes procedure summaries for all derivation trees of dimension 0, then they compute summaries for derivation trees of dimension 1 reusing the summaries computed for dimension 0 and so on.

Decomposition can be compared to refinement techniques based on automata [74, 77, 95] in which the aim is to eliminate sets of program traces that have been shown to be safe. Proof of the safety of a given dimension or dimensions of a set of clauses allows those dimensions to be eliminated, focusing the proof on the remaining dimensions. Our decomposition technique offers a very precise and practical approach to checking and eliminating infinite sets of traces.

7.7 EXPERIMENTAL RESULTS

We carried out an experiment on a set of 16 non-linear CHC verification problems taken from the repository[2] of software verification benchmarks. Our aim here is not to make a systematic comparison with other verification techniques; these are exploratory experiments to establish whether dimension-based decomposition is practical. The results are summarized in Table 7.1. Columns **Program**, **Result**, **Time** and **dim(k)** respectively represent a program, its verification result using our approach, time in seconds taken to generate the programs and solve it and a value of a proof decomposition parameter k.

2 https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/Eldarica/RECUR/

For the safety check (the procedure *SAFE* in Algorithm 1) we use the verification procedure described in [94] which uses abstract interpretation over the domain of convex polyhedra, with a timeout of 5 minutes. The symbol "-" in Table 7.1 denotes that we were unable to solve these problems within the given time. Our approach solves 14 out of 16 problems with an average time of 4 seconds (over the solved problems). Our previous approach based on refinement with finite tree automata described in [95] solves 1 additional problem, that is, *triple* than our current approach. These examples were also run on QARMC [65] which solves all the problems (much faster).

Most of the problems are solved when we decompose the proof with the value of $k = 0$. This indicates that separating the proofs for linear programs eases the verification task. The splitting induced as a result of separating a set of traces has an effect on delaying join and widening operations during convex polyhedra analysis which increases its precision.

7.8 CONCLUSION AND FUTURE WORK

We presented a program transformation approach to Horn clause verification using the notion of *tree dimension* to decompose the verification problem by separating dimensions. We presented one algorithm based on this idea which yielded preliminary results on set of non-linear Horn clause verification benchmarks, showing that the approach is feasible and this transformation is useful both for proving safety of a program as well as for finding bugs.

Other ideas of program verification based on tree-dimension are worth investigating, including proof by induction based on tree dimension, and further investigation of proof strategies that could exploit knowledge of dimension bounds (such as those discussed in Section 7.5).

Although it is formulated in the context of Datalog, it is known from Afrati *et al.* [3] that a set of CHC of bounded dimension can be turned into an equivalent set of linear CHC. The exact complexity of their procedure is still open.

8

SOLVING NON-LINEAR HORN CLAUSES USING A LINEAR HORN CLAUSE SOLVER

With John P. Gallagher and Pierre Ganty

Abstract
In this chapter we show that checking satisfiability of a set of non-linear Horn clauses (also called a non-linear Horn clause program) can be achieved using a solver for linear Horn clauses. We achieve this by interleaving a *program transformation* with a satisfiability checker for linear Horn clauses (also called a *solver* for linear Horn clauses). The program transformation is based on the notion of *tree dimension*, which we apply to a set of non-linear clauses, yielding a set whose derivation trees have bounded dimension. Such a set of clauses can be linearised. The main algorithm then proceeds by applying the linearisation transformation and solver for linear Horn clauses to a sequence of sets of clauses with successively increasing dimension bound. The approach is then further developed by using a solution of clauses of lower dimension to (partially) linearise clauses of higher dimension. We constructed a prototype implementation of this approach and performed some experiments on a set of verification problems, which shows some promise.

Keywords: Horn clause linearisation, linear Horn clause solver, dimension bounded Horn clauses, Horn clause solving.

8.1 INTRODUCTION

Many software verification problems can be reduced to checking satisfiability of a set of Horn clauses (the *verification conditions*). In this chapter we propose an approach for checking satisfiability of a set of non-linear Horn clauses (clauses whose body contains more than one non-constraint atom) using a linear Horn clause solver. A *program transformation* based on the notion of *tree dimension* is applied to a set of non-linear Horn clauses; this gives a set of clauses that can be linearised and then solved using a *linear solver* for Horn clauses. This combination of dimension-bounding, linearisation and then solving with a linear solver is repeated for successively increasing dimension. The dimension of a tree is a measure of its non-linearity – for example a linear tree (whose nodes have at most one child) has dimension zero while a complete binary tree has dimension equal to its height.

A given set of Horn clauses P can be transformed into a new set of clauses $P^{\leqslant k}$, whose derivation trees are the subset of P's derivation trees with dimension at most k. It is known that $P^{\leqslant k}$ can be transformed to a linear set of clauses preserving satisfiability; hence if we can find a model of the linear set of clauses then the original clauses $P^{\leqslant k}$ also have a model. Furthermore, $P^{\leqslant k}$ is included in $P^{\leqslant k+1}$. This allows reusing the solution of $P^{\leqslant k}$ to solve $P^{\leqslant k+1}$.

```
c1. fib(A, B):- A>=0,  A=<1, B=A.
c2. fib(A, B) :- A > 1, A2 = A - 2, fib(A2, B2),
                 A1 = A - 1, fib(A1, B1), B = B1 + B2.
c3. false:- A>5, fib(A,B), B<A.
```

Figure 8.1: Example CHCs Fib defining the Fibonacci function.

This motivated us to use this specific construction based on *tree dimension* to underapproximate a set of Horn clauses.

The algorithm terminates with success if a model (solution) M of $P^{\leqslant k}$ is also a model (after appropriate translation of predicate names) of P. However if M is not a solution of P, then we proceed to generate $P^{\leqslant k+1}$ and repeat the procedure. The algorithm terminates if $P^{\leqslant k}$ is shown to be unsatisfiable (unsafe) for some k, since this implies that P is also unsatisfiable.

A more sophisticated version of the algorithm attempts to use the model M of $P^{\leqslant k}$ to (partially) linearise $P^{\leqslant k+1}$. We can exploit the model of $P^{\leqslant k}$ in the following way; if $P^{\leqslant k+1}$ has a counterexample that does not use the (approximate) solution M for $P^{\leqslant k}$, then P is unsatisfiable. We continue this process successively for increasing values of k until we find a solution or a counterexample to P, or until resources are exhausted.

As an example program, we consider a set of constrained Horn clauses P in Figure 8.1 which defines the Fibonacci function. It is an interesting problem since its derivations are trees whose dimensions depend on an input argument. The last clause represents a property of the Fibonacci function expressed as an integrity constraint.

We have made a prototype implementation of this approach and performed some experiments on a set of software verification problems, which shows some promise. The main contributions of this chapter are as follows.

1. We present a linearisation procedure for dimension-bounded Horn clauses using partial evaluation (Section 8.3).
2. We give an algorithm for solving a set of non-linear Horn clauses using a linear Horn clause solver (Section 8.4).
3. We demonstrate the feasibility of our approach in practice applying it to non-linear Horn clause problems (Section 8.5).

8.2 PRELIMINARIES

A constrained Horn clause (CHC) is a FOL formula of the form $p(X) \leftarrow \mathcal{C}, p_1(X_1), \ldots, p_k(X_k)$ ($k \geqslant 0$) (using Constraint Logic Programming syntax), where $\mathcal{C}$ is a conjunction of *constraints* with respect to some constraint theory, X_i, X are (possibly empty) vectors of distinct *variables*, $p_1, \ldots, p_k, p$ are *predicate symbols*, $p(X)$ is the *head* of the clause and $\mathcal{C}, p_1(X_1), \ldots, p_k(X_k)$ is the *body*. An atomic formula, or simply *atom*, is a formula $p(t)$ where p is a non-constraint predicate symbol and t a tuple of arguments. Atoms are sometimes written as A, B or H, possibly with sub- or superscripts.

A clause is called *non-linear* if it contains more than one atom in the body, otherwise it is called *linear*. A set of Horn clauses P is called linear if P only contains linear clauses, otherwise it is called non-linear. *Integrity constraints* are a special kind of Horn clauses whose head is *false* where *false* is always interpreted as FALSE. A set of Horn clauses is sometimes called a *(constraint logic) program*.

An *interpretation* of a set of CHCs is represented as a set of *constrained facts* of the form $A \leftarrow \mathcal{C}$ where A is an atomic formula $p(Z)$ where Z is a tuple of distinct variables and $\mathcal{C}$ is a constraint over Z with respect to some constraint theory. An interpretation that makes each clause in P TRUE is called a *model* of P. We say a set of Horn clause P (including integrity constraints) is safe (solvable) iff it has a model. In some works e.g. [15, 126], a model is also called a *solution* and we use them interchangeably here.

A labeled tree $c(t_1, \ldots, t_k)$ ($k \geqslant 0$) is a tree whose nodes are labeled by identifiers, where c is the label of the root and $t_1, \ldots, t_k$ are labeled trees, the children of the root.

Definition 8.1 (Tree dimension (adapted from [45])). *Given a labeled tree* $t = c(t_1, \ldots, t_k)$*, the tree dimension of* t *represented as* $dim(t)$ *is defined as follows:*

$$dim(t) = \begin{cases} 0 & \text{if } k = 0 \\ \max_{i \in [1..k]} dim(t_i) & \text{if there is a unique maximum} \\ \max_{i \in [1..k]} dim(t_i) + 1 & \text{otherwise} \end{cases}$$

Given a set of Horn clauses, we associate with each clause $p(X) \leftarrow \mathcal{C}, p_1(X_1), \ldots, p_k(X_k)$ a unique identifier c whose *arity* is k.

Labelled trees can represent Horn clause derivations, where node labels are clause identifiers.

Definition 8.2 (Trace tree). *A* trace tree *for an atom* A *in a set of Horn clauses* P *is a labelled tree* $c(t_1, \ldots, t_k)$ *if* c *is a clause identifier for a clause* $A \leftarrow \mathcal{C}, A_1, \ldots, A_k$ *in* P *(with variables suitably renamed) and* $t_1, \ldots, t_k$ *are trace trees for* $A_1, \ldots, A_k$ *in* P *respectively.*

There is a one-one correspondence between *trace trees* and derivation trees of Horn clauses up to variable renaming. Thus when we speak about the dimension of a Horn clause derivation, we refer to the dimension of its corresponding *trace tree*.

Using the clauses shown in Figure 8.1 along with their identifiers, Figure 8.2 (a) shows a *trace tree* $t = c_3(c_2(c_2(c_1, c_1), c_1))$ and Figure 8.2 (b) shows its tree dimension. It can be seen that $dim(t) = 1$.

To make the chapter self contained, we describe the transformation to produce a dimension-bounded set of clauses. Given a set of CHCs P and $k \in \mathbb{N}$, we split each predicate p occurring in P into the predicates $p^{\leqslant d}$ and $p^{=d}$ where $d \in \{0, 1, \ldots, k\}$. An atom with predicate $p^{\leqslant d}$ or $p^{=d}$ is denoted $H^{\leqslant d}$ or $H^{=d}$ respectively. Such atoms have derivation trees of dimension at most d and exactly d respectively.

Definition 8.3 (At-most-k-dimension program $P^{\leqslant k}$). *Let* P *be a set of CHCs.* $P^{\leqslant k}$ *consists of the following clauses (adapted from [121, 99]):*

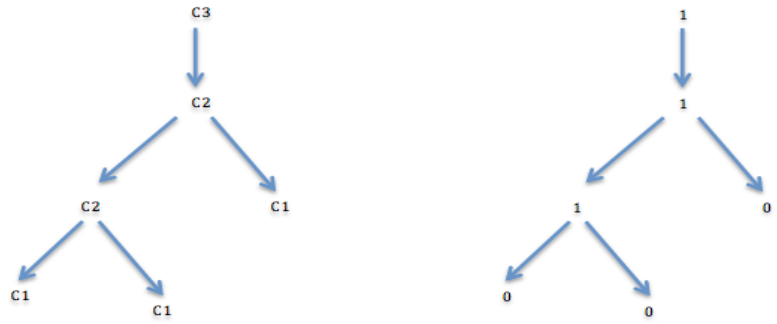

Figure 8.2: (a) a *trace tree* and (b) its tree dimension.

1. *Linear clauses:*

   If $H \leftarrow \mathcal{C} \in P$, *then* $H^{=0} \leftarrow \mathcal{C} \in P^{\leqslant k}$.

   If $H \leftarrow \mathcal{C}, B_1 \in P$ *then* $H^{=d} \leftarrow \mathcal{C}, B_1^{=d} \in P^{\leqslant k}$ *for* $0 \leqslant d \leqslant k$.

2. *Non-linear clauses:*

   If $H \leftarrow \mathcal{C}, B_1, B_2, \ldots, B_r \in P$ *with* $r > 1$ *and one of the following holds:*

   - *For* $1 \leqslant d \leqslant k$, *and* $1 \leqslant j \leqslant r$*:*

     Set $Z_j = B_j^{=d}$ *and* $Z_i = B_i^{\leqslant d-1}$ *for* $1 \leqslant i \leqslant r \wedge i \neq j$. *Then:* $H^{=d} \leftarrow \mathcal{C}, Z_1, \ldots, Z_r \in P^{\leqslant k}$.

   - *For* $1 \leqslant d \leqslant k$, *and* $J \subseteq \{1, \ldots, r\}$ *with* $|J| = 2$*:*

     Set $Z_i = B_i^{=d-1}$ *if* $i \in J$ *and* $Z_i = B_i^{\leqslant d-1}$ *if* $i \in \{1, \ldots, r\} \setminus J$. *If all* Z_i *are defined, i.e.,* $d \geqslant 2$ *if* $r > 2$, *then:* $H^{=d} \leftarrow \mathcal{C}, Z_1, \ldots, Z_r \in P^{\leqslant k}$.

3. ϵ*-clauses:*

   $H^{\leqslant d} \leftarrow H^{=e} \in P^{\leqslant k}$ *for* $0 \leqslant d \leqslant k$, *and every* $0 \leqslant e \leqslant d \leqslant k$.

$P^{\leqslant k}$ is also called the k-dimension-bounded program corresponding to P. When the value of k is not important, any program generated using the Definition 8.3 is called a dimension-bounded program. The relation between P and its k-dimensional program is given by in the Proposition 8.1 where $\models$ is the usual "logical consequence" operator.

Proposition 8.1 (Relation between P and $P^{\leqslant k}$)**.** *Let* P *be a program and* $P^{\leqslant k}$ *(*$k \geqslant 0$*) be its* k*-dimension-bounded program. Let* $p(t)$ *be an atom where* p *is a predicate of* P *and* $p^\star(t)$ *(*$\star \in \{=, \leqslant\}$*) be an atom where* $p^\star$ *is a predicate of* $P^{\leqslant k}$*. Then we have:* $P^{\leqslant k} \models p^\star(t) \implies P \models p(t)$.

In other words, Proposition 8.1 says that the set of facts that can be derived from $P^{\leqslant k}$ is a subset of the set of facts that can be derived from P, taking the predicate renaming into account. In this sense $P^{\leqslant k}$ is an *under-approximation* of P. In particular, if $P^{\leqslant k} \models \texttt{false}^\star$ then $P \models \texttt{false}$.

Let S be an interpretation of a dimension-bounded set of clauses $P^{\leqslant k}$. That is, S is a set of constrained facts of the form $H^{=d} \leftarrow \mathcal{C}$ or $H^{\leqslant d} \leftarrow \mathcal{C}$. An interpretation of P is constructed from S as follows.

```
%linear clauses
1. fib(0)(A,B) :- A>=0, A=<1, B=A.
2. false(0) :- A>5, B<A, fib(0)(A,B).
%epsilon-clauses
3. false[0] :- false(0).
4. fib[0](A,B) :- fib(0)(A,B).
```

Figure 8.3: $\text{Fib}^{\leqslant 0}$: at-most-0-dimension program of Fib.

```
fib(0)(A,B) :- B=A,  A=<1, A>=0.
fib(1)(A,B) :-  B=F+D, C=A-2,
               E=A-1,  A>1, fib[0](E,F), fib(1)(C,D).
fib(1)(A,B) :-  B=F+D, C=A-2, E=A-1,
               A>1, fib[0](C,D), fib(1)(E,F).
fib(1)(A,B) :-  B=F+D,  C=A-2,  E=A-1,
               A>1, fib(0)(C,D), fib(0)(E,F).
false(1) :-  B<A,  A>5,  fib(1)(A,B).
false(0) :-  B<A,  A>5, fib(0)(A,B).
false[1] :-  false(1).
false[1] :- false(0).
false[0] :-  false(0).
fib[1](A,B) :-  fib(1)(A,B).
fib[1](A,B) :- fib(0)(A,B).
fib[0](A,B) :- fib(0)(A,B).
```

Figure 8.4: $\text{Fib}^{\leqslant 1}$: at-most-1-dimension program of Fib.

Definition 8.4 ($S^{\uparrow P^{\leqslant k}}$: an interpretation of P constructed from an interpretation of $P^{\leqslant k}$)**.** *Let* S *be an interpretation of* $P^{\leqslant k}$*. Then* $S^{\uparrow P^{\leqslant k}}$ *is the following set of constrained facts.*

$$S^{\uparrow P^{\leqslant k}} = \{p(X) \leftarrow \bigvee\{\mathcal{C} \mid p^{=d}(X) \leftarrow \mathcal{C} \in S \vee p^{\leqslant d}(X) \leftarrow \mathcal{C} \in S\} \mid \ d \in \{0...k\} \text{ and } p : \text{ pred. in } P\}$$

The set $S^{\uparrow P^{\leqslant k}}$ *is a disjunctive interpretation of* P *where the interpretation of* p *is the disjunction of the interpretations of the corresponding dimension-bounded versions of* p *in* $P^{\leqslant k}$*.*

The at-most-0-dimension program of Fib in Figure 8.1 is depicted in Figure 8.3. In textual form we represent a predicate $p^{\leqslant k}$ by `p[k]` and a predicate $p^{=k}$ by `p(k)`. The at-most-1-dimension program of Fib in Figure 8.1 is depicted in Figure 8.4. Note that 0-dimension program is included in 1-dimension program. In general, all the clauses in $P^{\leqslant k}$ are also in $P^{\leqslant k+1}$. This provides a basis for an iterative strategy for a bounded set of Horn clauses. Since some programs have derivation trees of unbounded dimension, trying to verify a property for its increasing dimension separately is not a practical strategy. It only becomes a viable approach if a solution of $p^{\leqslant k}$ for some $k \geqslant 0$ is general enough to hold for all dimensions of P.

8.3 LINEARISATION STRATEGIES FOR DIMENSION-BOUNDED SET OF HORN CLAUSES

In this section, we present linearisation strategies for set of clauses of bounded dimension. It is known [3] that a dimension-bounded set of clauses can be linearised, preserving satisfiability. In this section we describe a practical technique for linearisation, based on partial evaluation of an interpreter.

8.3.1 *Linearisation based on partial evaluation*

Partial evaluation (PE) has been studied for a variety of languages including logic programs [89, 58, 110, 91, 115]. We follow the pattern of transforming a program (a set of Horn clauses) by specialising an interpreter for that program [51, 91]. Let PE be a partial evaluator, I an interpreter and P an object program. Then the partial evaluation of I with respect to P, denoted $PE(I, P)$, represents the "compilation" of P using the semantics given by I.

We first write an interpreter for Horn clause programs, which is also written as a set of Horn clauses. Given a (possibly empty) conjunction of atoms (called a *goal*) the interpreter constructs a derivation, implementing a standard left-to-right, depth-first search. In the interpreter predicate `solve(Gs)`, `Gs` is the goal, represented as a list of atoms. The basic step of the interpreter is represented by the clauses for `solve(Gs)` shown in Figure 8.5. If the conjunction is not empty, its first atom `G` is selected along with a matching Horn clause $G \leftarrow Cs, B$ in the program being interpreted, where Cs is a conjunction of constraints and B is a conjunction of atoms. This clause is represented by `hornClause(G,Cs,B)` in the interpreter. The body of the clause is conjoined with the remaining goal atoms, and the derivation continues with the new goal `Gs1`. If the conjunction is empty, the derivation is successful (second clause).

```
solve([G|Gs]) :-
   hornClause(G,Cs,B), solveConstraints(Cs), append(B,Gs,Gs1),
   solve(Gs1).
solve([]).
```

Figure 8.5: Depth-first interpreter for Horn clauses

To interpret a dimension-bounded set of clauses (say the bound is k), we use the fact that in all successful runs of the interpreter in which goals are selected in increasing order of dimension, the size of the conjunction of goals (that is, the length of the argument of `solve`) has an upper bound related to k. This bound is known as the *index* of the set of clauses and is given as $(i-1)*k+1$, where i is the maximum number of non-constraint atoms in the body of clauses [47]. Given this index, we can augment the interpreter with a check on the size of the conjunction, ensuring that it never exceeds the index. In addition, due to the requirement of increasing dimension in the selection of atoms, a left-to-right computation rule is not sufficient; therefore we permute the set of atoms in each clause body, since in at least one permutation the goals will be ordered by dimension. With these changes the interpreter remains complete for clauses of the given maximum index, at the possible cost of some redundancy in the search.

These additions result in the interpreter whose top level is shown in Figure 8.6. Let the interpreter predicate `solve(Gs,Index,L)` mean that the conjunction of goals `Gs` is to be solved, and `L`, `Index` are numbers representing the size of `Gs` and the maximum size of the stack of goals.

```
go(Index) :-
   solve([false],Index,1).
solve([G|Gs],Index,L) :-
   hornClause(G,Cs,B), solveConstraints(Cs),
   length(B,L1), L2 is L1+L-1, L2 =< Index,
   perm(B,B1), append(B1,Gs,Gs1),
   solve(Gs1,Index,L2).
solve([],_,_).
```

Figure 8.6: Interpreter for linearisation

PARTIAL EVALUATION OF THE INTERPRETER. Given a set of facts of the form `hornClause(G,Cs,B)` representing the Horn clauses to be linearised, and some value of `Index`, the interpreter can be partially evaluated. We use Logen [116] to perform the partial evaluation with respect to a call to `go(Index)`, which initiates a proof of the goal `false` (see first clause of interpreter). All interpreter computations are partially evaluated except for the calls to `solve(Gs,Index,L)` and the execution of constraints within the goal `solveConstraints(Cs)`. Furthermore Logen performs standard structure-flattening and predicate renaming operations, yielding a set of clauses of the form `solve_i(X) :- Cs, solve_j(Y)`, where `solve_i(X)` and `solve_j(Y)` are instantiations of
`solve(Gs,Index,L)` and `Cs` is a constraint. Thus the resulting clauses are linear, and furthermore preserve the meaning of the original clauses as given by the interpreter, by correctness of the partial evaluation procedure. The linearisation procedure is independent of the constraint theory underlying the clauses.

Proposition 8.2. *Let* P *be a program and* $P^{\leqslant k}$ *(*$k \geqslant 0$*) be its* k*-dimension-bounded program. Let* i *be the maximum number of atoms in clause bodies of* P*. Let* $\texttt{Index} = (i-1)*k+1$*. Let* P' *be a partial evaluation of the interpreter in Figure 8.6, with respect to* P *and the goal* `go(Index)`*. Then* $P^{\leqslant k} \models \mathit{false}^{\leqslant k}$ *iff* $P' \models \texttt{go(Index)}$*.*

Furthermore P' is linear if the partial evaluator follows the strategy described above. Combining Propositions 8.2 and 8.1, we conclude that $P' \models \texttt{go(Index)} \Rightarrow P \models \mathit{false}$.

Note that linearisation required partial evaluation of the `perm` predicate, giving a blow-up in program size related to the length of the clause bodies. This is further discussed at the end of Section 8.5.

8.3.2 *Obtaining linear over-approximations with a partial model*

First we note that the set of predicates in $P^{\leqslant k}$ is a subset of the set of predicates in $P^{\leqslant k+1}$. Given a model M for the predicates in $P^{\leqslant k}$, $P^{\leqslant k+1}$ can be linearised if we replace each occurrence of a predicate from $P^{\leqslant k}$ in the body of a clause in $P^{\leqslant k+1}$ with the corresponding constraint from the model M. The resulting set of clauses is linear since $P^{\leqslant k+1}$ contains at most one predicate in its body from $P^{\leqslant k+1}$ which is not in $P^{\leqslant k}$. Furthermore if $P^{\leqslant k+1}$ has a model then so does the set of clauses resulting from the replacement; the converse is however not the case since the model M represents an over-approximation of $P^{\leqslant k}$. An example is given in Section 8.4.

More generally, we can replace any subset of the occurrences of predicates from $P^{\leqslant k}$ in $P^{\leqslant k+1}$. We summarise this in the following lemma.

Lemma 8.1 (Linear over-approximation)**.** *Let M be a model of the predicates in $P^{\leqslant k}$, represented by a set of "constrained facts" $p(X) \leftarrow \mathcal{C}$ where p is a predicate in $P^{\leqslant k}$. Let P' be any set of clauses obtained from $P^{\leqslant k+1}$ by replacing some of the occurrences of predicates $p(X)$ from $P^{\leqslant k}$ in the bodies of clauses in $P^{\leqslant k+1}$ with their corresponding interpretation $\mathcal{C}$ in M. Then*

1. *If $P^{\leqslant k+1}$ has a model then so does P';*
2. *If P' contains no predicate from $P^{\leqslant k}$, then P' is linear.*

8.4 ALGORITHM FOR SOLVING SETS OF NON-LINEAR HORN CLAUSES

A basic procedure for solving a set of non-linear Horn clauses using a linear Horn clause solver is presented in Algorithms 8.1 and 8.2. We use the term "linear solver" for linear Horn clause solver for brevity. The main procedure **SOLVE(*P*)** takes a set of non-linear Horn clauses P as input and outputs (upon termination) *(safe, solution)* if P is solvable or *(unsafe, counterexample)* otherwise. We represent a counterexample as a *trace tree*. For a linear program it corresponds to a sequence of clauses used to derive a counterexample.

Definition 8.5. ($S_{|t}$) *Let S be an interpretation of a set of Horn clauses P. Let t be any trace tree for some atom A in P (Definition 8.2) and let A_t be the set of heads of clauses with identifiers in t. Then $S_{|t}$ is defined to be the set*

$$S_{|t} = \{(H \leftarrow \mathcal{C}) \mid (H \leftarrow \mathcal{C}) \in S \wedge H \notin A_t\}.$$

Informally, the derivation corresponding to t does not use any predicate interpreted by $S_{|t}$. This notion is used in Algorithm 8.2.

Algorithm 8.2 is an extended version of Algorithm 8.1, which uses the solution for $P^{\leqslant k}$ to help to linearise $P^{\leqslant k+1}$ and also allows a more refined termination condition based on whether or not the solution for $P^{\leqslant k}$ is used in constructing a counterexample for $P^{\leqslant k+1}$.

The procedures make use of several sub-procedures which will be described next.

8.4.1 *Components of the algorithm*

- **KDIM(*P,k*)**: produces an at-most-k-dimension program $P^{\leqslant k}$ (Definition 8.3). By definition, $P^{\leqslant k}$ is linear for $k = 0$. For our example program presented in Figure 8.1, $Fib^{\leqslant 0}$ is shown in Figure 8.3 which is linear since there is at-most one non-constraint atom in the body of each clause.

- **SOLVE_LINEAR(*P′*)**: solves a set of linear Horn clauses P'. We assume the following about a linear solver: (i) if it terminates on P', then it returns either *safe* and a *solution* or *unsafe* and a *counterexample*; (ii) it is sound, that is, if it returns a *solution* S for P' then P' has a model and S is a solution (model) of P'; if it returns *unsafe* and a counterexample *cEx* then P' is unsafe and *cEx* is a witness. In our setting (Algorithms 8.1 and 8.2), P' corresponds to a linearised version of $P^{\leqslant k}$ for some P and $k \geqslant 0$. For technical reasons, the top level predicate $false^{=k}$ of $P^{\leqslant k}$ if any, is renamed to *false* before passing to a linear solver.

  In essence, any Horn clause solver which complies with our assumption, for example QARMC [65], Convex polyhedral analyser [96], ELDARICA [80] etc. can be used in a black-box fashion but in this chapter, we make use of a solver described in [96], which is based on abstract interpretation [32] over the domain of convex polyhedra [33] but without refinement using finite tree automata. The solver produces the following solution for the program in Figure 8.3. We can check it is in fact a solution (model).

```
fib(0)(A,B) :- [-A>= -1,A>=0,B=A].
fib[0](A,B) :- [-A>= -1,A>=0,B=A].
false[0] :- <>. % <>  means that there is no model for false[0],
%so we can discard it
```

- **LINEARISE(P,k,S)** generates a linear set of clauses from $P^{\leqslant k}$ and an interpretation S for $P^{\leqslant k}$. Let S be a set of constrained facts of the form $p(X) \leftarrow \mathcal{C}$, where p is a predicate from $P^{\leqslant k}$, the procedure replaces every clause from $P^{\leqslant k}$ with head $p(X)$ by $p(X) \leftarrow \mathcal{C}$. This produces a set of clauses say P'. Then the procedure **LINEARISE_PE(**P',Index**)** is called, which is the linearisation procedure based on partial evaluation described in Section 8.3 where Index is a bound for the stack usage for linearising $P^{\leqslant k}$.

An excerpt from $Fib^{\leqslant 1}$ is shown below.

```
false(1) :- A>5, B<A,  fib(1)(A,B).
fib(1)(A,B) :- A>1, C=A-2, E=A-1, B=F+D,  fib(1)(C,D),  fib[0](E,F).
fib(0)(A,B) :- B=A,  A=<1, A>=0.
```

After reusing the solution obtained for $Fib^{\leqslant 0}$ and linearising, we obtain the following set of linear clauses.

```
false(1) :-  A>5, B<A, fib(1)(A,B).
fib(1)(A,B) :-  -A>= -2, A>1, A-C=2,  B-D=1, fib(1)(C,D).
```

Continuing to run our algorithm, the following solution obtained for $\texttt{Fib}^{\leqslant 2}$ becomes a solution for the program in Figure 8.1 (the original program) and the algorithm terminates.

```
fib(0)(A,B) :- [-A>= -1,A>=0,B=1].
fib[0](A,B) :- [-A>= -1,A>=0,B=1].
fib(1)(A,B) :- [A>=2,A+ -B=0].
fib[1](A,B) :- [A+ -B>= -1,B>=1,-A+B>=0].
fib(2)(A,B) :- [A>=4,-2*A+B>= -3].
fib[2](A,B) :- [A>=0,B>=1,-A+B>=0].
```

Algorithm 8.1: Algorithm for solving a set of Horn clauses

1 Procedure **SOLVE(*P*)**
  Input: A set of CHCs P
  Output: *(safe, solution), (unsafe, cex)*
2 $k \leftarrow 0$;
3 $P' \leftarrow$ **LINEARISE**$(P, k, \emptyset)$;
4 $(status, Result) \leftarrow$ **SOLVE_LINEAR(*P'*)** ; `/* Result is a solution or a cex */`
5 **if** *status* $=$ *safe* **then**
6 | **if** *(*$Result^{\uparrow P^{\leqslant k}}$ *is a solution of* P*)* **then return** $(safe, Result^{\uparrow P^{\leqslant k}})$;
7 | ;
8 | $k \leftarrow k + 1$;
9 **else**
10 | **return** $(unsafe, Result)$; `/* Result is a cex */`
11 **goto** 3;

8.4.2 *Reuse of solutions, refinement and linearisation*

Algorithm 8.2 solves non-linear Horn clauses P in essentially the same way as Algorithm 8.1, but incorporates a *refinement* phase in the case that the linear solver finds a counterexample. This counterexample possibly uses some of the model of the lower-dimension predicates S, in which case it is not certain whether it is a false alarm or a real counterexample. If the counterexample did use some of the predicate solutions from S, then we discard those solutions (Algorithm 8.2, line 12) and return to the linearisation step. If the counterexample does not use any predicate solutions from S, then it is a real counterexample (Algorithm 8.2, line 12). We will clarify this with an example program (linear for simplicity) shown below.

```
c1. false:- X=0, p(X).
c2. false:- q(X).
c3. p(X):- X>0.
c4. q(X):-X=0.
```

Algorithm 8.2: Algorithm for solving a set of Horn clauses, with reuse of lower dimension solutions

1 Procedure **SOLVE(*P*)**
  Input: A set of CHCs $\mathtt{P}$
  Output: *(safe, solution), (unsafe, cex)*
2 $\mathtt{k} \leftarrow 0$;
3 $\mathtt{S} \leftarrow \emptyset$;
4 $\mathtt{P}' \leftarrow$ **LINEARISE**$(\mathtt{P}, \mathtt{k}, \mathtt{S})$;
5 $(status, Result) \leftarrow$ **SOLVE_LINEAR(*P′*)** ; `/* Result is a solution or a cex */`
6 **if** *status* $=$ *safe* **then**
7 **if** *(*$Result^{\uparrow \mathtt{P}^{\leqslant \mathtt{k}}}$ *is a solution of* $\mathtt{P}$*)* **then return** $(safe, Result^{\uparrow \mathtt{P}^{\leqslant \mathtt{k}}})$;
8 ;
9 $\mathtt{k} \leftarrow \mathtt{k} + 1$;
10 $\mathtt{S} \leftarrow Result$;
11 **else**
12 **if** $\mathtt{S} = \mathtt{S}_{|Result}$ **then return** $(unsafe, Result)$;; `/* Result is a linear cex */`
13
14 $\mathtt{S} \leftarrow \mathtt{S}_{|Result}$; `/*` $\mathtt{S}_{|Result}$`: Definition 8.5 */`
15 **goto** 4;

Algorithm 8.3: Algorithm for linearising a set of clauses

1 Procedure **LINEARISE(*P, k, S*)**
  Input: A set of CHCs $\mathtt{P}$, an integer $\mathtt{k}$ and a set of constrained facts S
  Output: A linearised set of clauses $\mathtt{P}_{\mathtt{lin}}$
2 $\mathtt{P}^{\leqslant \mathtt{k}} \leftarrow$ **KDIM(*P,k*)** ; `/* Definition 8.3 */`
3 $\mathtt{P}' \leftarrow$ **SUBSTITUTE**$(\mathtt{P}^{\leqslant \mathtt{k}}, S)$; `/* substitute atoms of` $\mathtt{P}^{\leqslant \mathtt{k}}$ `with their interpretations from` S `*/`
4 $\mathtt{Index} \leftarrow (\mathtt{i} - 1) * \mathtt{k} + 1$; `/* where i is the maximal number of body atoms of` $\mathtt{P}$ `*/`
5 $\mathtt{P}_{\mathtt{lin}} \leftarrow$ **LINEARISE_PE(*P′, Index*)** ; `/* Section 8.3.1 */`
6 **return** $\mathtt{P}_{\mathtt{lin}}$

Suppose we have an approximate solution $S = \{p(X) : -true\}$ for the predicate $p(X)$. Using this solution, the above program is transformed into the following program.

```
c1. false:- X=0, p(X).
c2. false:- q(X).
c3. p(X):- true. (approximate solution)
c4. q(X):-X=0.
```

The trace $c_1(c_3)$ is a counterexample for this transformed program but not to the original program (since it uses an approximate solution for the predicate p). However the trace $c_2(c_4)$ is a counterexample for this program as well as to the original since it does not use any approximate solution for the predicates appearing in the counterexample.

A schematic overview of Algorithm 8.2 is shown in Figure 8.7. At each iteration of the *abstraction-refinement* loop, the at-most-k-dimension under-approximation of P is computed, then linearised and solved using a solver for linear Horn clauses.

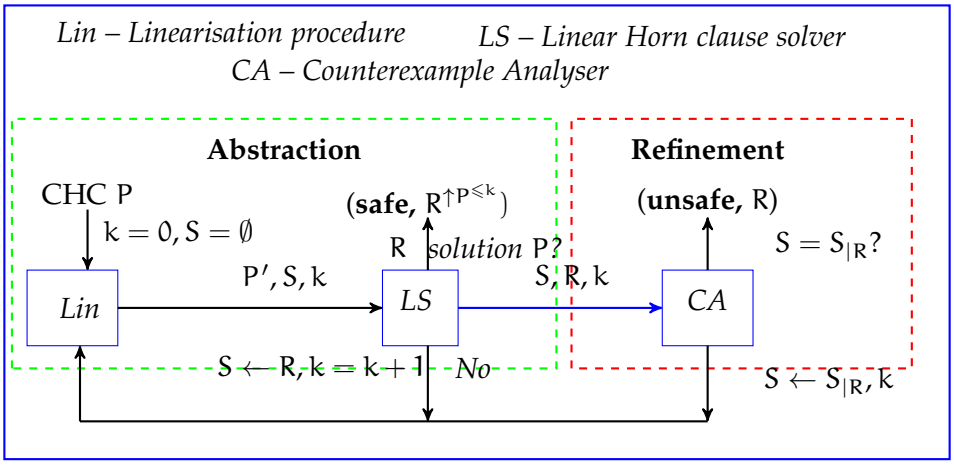

Figure 8.7: *Abstraction-refinement scheme for solving non-linear Horn clauses using a solver for linear Horn clauses.* P' *is a set of linear CHC obtained by linearising the at-most-*k*-dimension underapproximation,* $P^{\leqslant k}$*, of* P.

The soundness of Algorithms 8.1 and 8.2 is captured by Proposition 8.3.

Proposition 8.3 (Soundness). *If Algorithm 8.1 or 8.2 returns safe and a solution* S *for a set of clauses* P *then* P *is safe and* S *is in fact a solution of* P*; if it returns unsafe and a counterexample cEx then* P *is unsafe and cEx is a witness.*

Another property of the Algorithm 8.2 is that of progress, that is, the same counterexample does not arise more than once.

8.5 EXPERIMENTAL RESULTS

We made a prototype implementation of Algorithm 8.2 in the tool called *LHornSolver*[1]. It uses the solver described in [96] without refinement as a linear Horn clause solver, but it would

1 https://github.com/bishoksan/LHornSolver

have been ideal to use a specialised linear Horn clause solver. We did not find any such solver which fulfills our purpose. *LHornSolver* is written in Ciao Prolog [78] and is interfaced with the Parma Polyhedra Library [5] and the Yices SMT solver [44] for handling constraints. Then we carried out an experiment on a set of 44 non-linear CHC verification problems taken from the repository[2] of software verification benchmarks, the recursive category of SV-COMP[3] [13] and the tool QARMC. The experiments were run on a MAC computer running OS X on 2.3 GHz Intel core i7 processor and 8 GB memory. The benchmarks that we use in the experiments are not beyond the capabilities of existing solvers, but they are challenging. These programs are first translated to Prolog syntax using the tools ELDARICA [80] and SeaHorn [71]. Our aim with these experiments is to explore: (1) whether using a linear solver for non-linear problem solving is practical; (2) the relationship between the solvability of a problem and its dimension; and (3) how the current results compare with the results using the state of the art non-linear Horn clause verification tool (in our case RAHFT [96]). The results are summarized in Table 8.1.

In the table *Program* represents a program, *Safety* represents a verification result, *#iter.* and *Time (s)* successively represent the number of refinement iterations and the time in seconds need to solve a program using both RAHFT and *LHornSolver*. It is to note that the underlying abstract interpreter, that is, the convex polyhedral analyser (CPA) is the same for both RAHFT and *LHornSolver* but *LHornSolver* uses it to solve linear Horn clauses though the CPA is not optimised for linear problems. The column *#iter.* for *LHornSolver* represents a value of k for which a solution of $P^{\leqslant k}$ (under-approximation) of a set of clauses P becomes a solution for P or $P^{\leqslant k}$ becomes unsafe. The symbol "?" means that the result is unknown within the given time bound. The result "safe" means that the program is safe (solvable) and "unsafe" means it is unsafe.

LHornSolver solves 27 out of 44 (about 61%) problems within a second. In most of these problems, a solution of an under approximation ($P^{\leqslant k}$) becomes a solution for the original program or $P^{\leqslant k}$ becomes unsafe for a fairly small value of k (1 or 2). This suggests that the solvability of a problem is shallow with respect to its dimension. This demonstrates the feasibility of solving a set of non-linear Horn clauses using a solver for linear Horn clauses.

In contrast, RAHFT solves all the problem. The difference in results maybe due to the following reason: the linear solver that is used in *LHornSolver* is the CPA (without refinement in contrast to [96]). The solver terminates but produces *false alarms*. If we use CPA with refinement as in [96], then we lose predicate names (due to program transformation), so the solution or counterexamples produced by the tool do not correspond to the original program (it is very hard to keep track of the changes). This hinders the reuse of solution from lower dimension to linearise program of higher dimension or refine it using the counterexample trace. Other solvers which don't modify the programs but produce solutions or counterexamples can be used as a linear solver in principle and we leave it for the future work. Another disadvantage of using CPA is that, if it cannot solve a linear program, then it emits an abstract trace which is checked for feasibility. If it is spurious then LHornSolver returns with *unknown* (in principle we can refine the program but the refinement will have the problem as men-

2 https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/Eldarica/RECUR/
3 http://sv-comp.sosy-lab.org/2015/benchmarks.php

	RAHFT			***LHornSolver***		
Program	**Safety**	**# iter.**	**Time (s)**	**Safety**	**#iter.**	**Time (s)**
Addition03_false-unreach	safe	2	< 1	?	?	?
McCarthy91_false-unreach	unsafe	0	< 1	?	?	?
addition.nts.pl	safe	0	< 1	safe	1	< 1
bfprt.nts.pl	safe	0	< 1	safe	2	4
binarysearch.nts.pl	safe	0	< 1	safe	1	1.1
countZero.nts.pl	safe	0	< 1	safe	1	< 1
eq.horn	unsafe	0	< 1	unsafe	2	< 1
fib.pl	safe	0	< 1	?	?	?
identity.nts.pl	safe	0	< 1	safe	1	< 1
merge.nts.pl	safe	0	< 1	safe	1	1.7
palindrome.nts.pl	safe	0	< 1	safe	1	< 1
parity.nts.pl	unsafe	1	< 1	?	?	?
remainder.nts.pl	unsafe	0	< 1	unsafe	1	< 1
revlen.pl	safe	0	< 1	safe	1	< 1
running.nts.pl	unsafe	1	< 1	?	?	?
sum_10x0_false-unreach	unsafe	10	10	?	?	?
sum_non_eq_false-unreach	unsafe	0	< 1	?	?	?
suma1.horn	unsafe	0	< 1	unsafe	1	< 1
suma2.horn	unsafe	0	< 1	unsafe	2	< 1
summ_SG1.r.horn	safe	0	< 1	?	?	?
summ_SG2.r.horn	safe	8	78	?	?	?
summ_SG3.horn	safe	0	< 1	safe	1	< 1
summ_b.horn	safe	2	1.7	?	?	?
summ_binsearch.horn	safe	1	3	?	?	?
summ_cil.casts.horn	safe	0	< 1	safe	1	< 1
summ_formals.horn	safe	0	< 1	safe	1	< 1
summ_g.horn	safe	0	< 1	?	?	?
summ_globals.horn	safe	0	< 1	safe	1	< 1
summ_h.horn	safe	0	< 1	safe	2	< 1
summ_local-ctx-call.horn	safe	0	< 1	safe	1	< 1
summ_locals.horn	safe	0	< 1	?	?	?
summ_locals2.horn	safe	0	< 1	safe	1	< 1
summ_locals3.horn	safe	0	< 1	safe	1	< 1
summ_locals4.horn	safe	0	< 1	safe	2	2.2
summ_mccarthy2.horn	safe	3	5	?	?	?
summ_multi-call.horn	safe	0	< 1	safe	1	< 1
summ_nested.horn	safe	0	< 1	safe	1	< 1
summ_ptr_assign.horn	safe	0	< 1	safe	1	< 1
summ_recursive.horn	safe	0	< 1	?	?	?
summ_rholocal.horn	safe	0	< 1	safe	1	< 1
summ_rholocal2.horn	safe	0	< 1	safe	1	< 1
summ_slicing.horn	safe	0	< 1	?	?	?
summ_summs.horn	safe	0	< 1	?	?	?
summ_typedef.horn	safe	0	< 1	safe	1	< 1
summ_x.horn	safe	0	< 1	?	?	?
Average		0.64	2.3		1.185	< 1

Table 8.1: Experimental results on non-linear CHC verification problems with a timeout of 5 minutes.

tioned above). So it is highly unlikely that the trace picked by the tool non-deterministically results to be a real counterexample. We noticed in our experiments that the trace picked was spurious most of the times and *LHornSolver* immediately returned "unknown" answer before the timeout. This also explains why solving time of *LHornSolver* is less than that of RAHFT.

The interpreter described in Figure 8.6 computed a *permutation* of the atoms in a clause body; partial evaluation of the permutation procedure can cause a blow-up of the size of the linearised program, relative to the number of atoms in clause bodies. During our experiment we found that the maximum number of atoms in the bodies of the clauses in our benchmark programs was 5 and the value of k was relatively small ($k = 0 \ldots 2$). The permutation procedure can be avoided if we first generate an at-most-k-dimension program whose body atoms are ordered by increasing dimension. This needs unfolding of the ϵ-clauses, since atoms whose predicate is $p^{\leqslant d}$ cannot be ordered directly; only atoms with predicates of the form $p^{=d}$can be ordered. We have not yet evaluated the trade-offs in these two approaches.

8.6 RELATED WORK

In the world of Horn clause solvers, after fixing a constraint theory, we can distinguish solvers depending on whether they can handle general non-linear Horn clauses or not. A majority of solvers [71, 64, 136, 126, 96] handle non-linear Horn clauses but there are notable exceptions like VeriMAP [37] or Sally[4]. For both VeriMAP and Sally, their underlying reasoning engine handles only linear Horn clauses which restricts, in principle, their applicability. The developers of VeriMap claim that linearity is not a restriction in principle [40], but in practice we cannot just get away with non-linear problems because of their ubiquity and one has to find a way to deal with them. Our contribution is to lift this restriction by allowing those tools to be applied on arbitrary sets of Horn clauses, linear or not, through a linearisation procedure that underapproximates the set of solutions. We give empirical evidence that this underapproximation often provides enough coverage to enable the verification of the original set of Horn clauses. To summarize, we allow solvers with restrictions to be applied on any input at the price of an under-approximation which often results in full coverage.

Our linearisation method based on partial evaluation described in Section 8.3.1 is related to the linearisation method based on *fold-unfold transformations* described by De Angelis *et al.* [40]. While their procedure transforms the target set of clauses directly, we transform an interpreter for the clauses using a generic partial evaluation procedure. Any clause transformation procedure could be formulated as a meta-program and partial evaluation applied to that program to yield the specified transformation. Thus neither approach offers any more power than the other. However the use of partial evaluation is arguably more flexible. The interpreter that is partially evaluated in our procedure is a standard interpreter for Horn clauses, modified with a bound on the size of goals, directly incorporating a general result that there is an upper bound on the size of goals in derivations with dimension-bounded programs. This provides a very generic starting point for the transformation with an explicit relation to the semantics of the clauses. A whole family of similar transformations could be formulated by varying

4 https://github.com/SRI-CSL/sally

the interpreter (for example using breadth-first search). The procedure in [40] is tailored to a restrictive setting where only goal clauses (integrity constraints) are non-linear and rest of the clauses are linear; correctness has to be established for that case. Unlike [40], we cannot transform clauses that do not have a solution in linear arithmetic to the clauses that have a solution.

Ganty, Iosif and Konečný [63] used the notion of *tree dimension* for computing summaries of procedural programs by underapproximating them. Roughly speaking, they compute procedure summaries iteratively, starting from the program behaviors captured by derivation trees of dimension 0. Then they reuse these summaries to compute summaries for program behaviors captured by derivation trees of dimension 1 and so on for 2, 3, etc. Kafle, Gallagher and Ganty [99] adapted the idea of dimension-based underapproximations to the setting of Horn clause systems. They gave empirical evidence supporting the thesis that for small values of the dimension the solutions are general enough to hold for every dimension. Their approach still required the use of general Horn-clause solvers capable of handling non-linear clauses. In this chapter, we lift this requirement and allow the use of solvers for linear clauses only. Moreover, we provide an abstraction refinement loop that enables the solutions for lower dimension to be reused when searching for solutions in higher dimension.

8.7 CONCLUSION AND FUTURE WORK

We presented an *abstraction-refinement* approach for solving a set of non-linear Horn clauses using an off-the-shelf linear Horn clause solver. It was achieved through a linearisation of a dimension bounded set of Horn clauses (which are known to be linearisable) using partial evaluation and the use of a linear Horn clause solver. Experiment on a set of non-linear Horn clause verification problems using our approach shows that the approach is feasible (a linear solver can be used for solving non-linear problems) and the solvability of a problem is shallow with respect to its dimension.

A linear set of clauses is essentially a transition system. Many tools exist whose input languages have a form such as C programs (without procedure calls), control flow graphs, Boogie programs, and such formalisms whose semantics is usually given as a transitions system. The results suggest that such tools could be applied to the verification of non-linear Horn clauses.

In the future, we plan to compare our results with the results from a specialised linear Horn clause solver like VeriMap and other non-linear Horn clause solvers. We also plan to experiment with different linearisation strategies for Horn clauses and study their effects in Horn clause verification.

9

RAHFT: A TOOL FOR VERIFYING HORN CLAUSES USING ABSTRACT INTERPRETATION AND FINITE TREE AUTOMATA

With John P. Gallagher and José F. Morales

Abstract

We present RAHFT (Refinement of Abstraction in Horn clauses using Finite Tree automata), an *abstraction refinement* tool for verifying safety properties of programs expressed as Horn clauses. The chapter describes the architecture, strength and weakness, implementation and usage aspects of the tool. RAHFT loosely combines three powerful techniques for program verification: (i) program specialisation, (ii) abstract interpretation, and (iii) trace abstraction refinement in a non-trivial way, with the aim of exploiting their strengths and mitigating their weaknesses through the complementary techniques. It is interfaced with an abstract domain, a tool for manipulating finite tree automata and various solvers for reasoning about constraints. Its modular design and customizable components allows for experimenting with new verification techniques and tools developed for Horn clauses.

Keywords: Horn clause solvers, abstraction-refinement, tree automata, program specialisation, abstract interpretation.

9.1 CONSTRAINED HORN CLAUSE VERIFICATION AND OUR APPROACH

A constrained Horn clause (CHC) is a first order predicate logic formula usually written in the form $p(X) \leftarrow \phi, p_1(X_1), ..., p_k(X_k)$ ($k \geqslant 0$) using Constraint Logic Programming (CLP) syntax, where ϕ is a first order logic formula (constraint) with respect to some background theory, X_i, X are (possibly empty) tuples of distinct variables, and $p_1, \ldots, p_k, p$ are predicate symbols. There is a distinguished predicate symbol *false* which is interpreted as FALSE. Clauses with *false* head are called *integrity constraints*. A set of CHCs is called a (CLP) program.

An interpretation of a set of CHCs P is a set of *constrained facts* of the form $A \leftarrow \phi$ where A is an atom and ϕ is a formula with respect to some background theory. An interpretation that satisfies each clause is called a *model* (a *solution* in some works [15, 126]). In Horn clause verification, *integrity constraints* represent the safety properties to be verified; other clauses represent the program's behaviours. The CHC verification problem is to check whether there exists a model of P.

Several verification tools have been developed for CHCs, including SeaHorn [70], QARMC [65], VeriMap [37], Convex polyhedral analyser [96], TRACER [85], ELDARICA [80], and Trace abstraction refinement tool [147]. They exploit either Formulation I or Formulation II for Horn clause verification.

```
false :- Y>X, l(X,Y).
l(X1,Y1) :- X1=X+Y,
            Y1=Y+1,
            l(X,Y).
l(X,Y) :- X=1, Y=0.
```

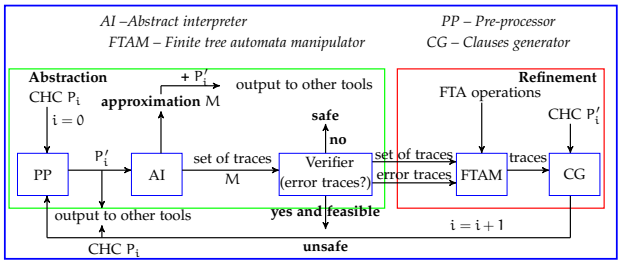

Figure 9.1: (a) Example program; (b) The architecture of RAHFT.

Formulation I (deductive): P has a model if and only if $P \nvdash \mathit{false}$ (*false* is not derivable from P). In CLP terminology, $P \vdash A$ if and only if the query $\leftarrow A$ succeeds in P. In this formulation it is sufficient to show that the query $\leftarrow \mathit{false}$ fails finitely or infinitely. Formulation I forms the basis of the tools described in [74, 147]. As the *minimal model* of P contains exactly the set of atoms that succeed [81], we have another formulation of the CHC verification problem [59].
Formulation II (model-based): P has a model if and only if $P \not\models \mathit{false}$. It forms the basis of tools based on *abstract interpretation*, *interpolation* or *predicate abstraction* [65, 70, 96].

The program in Figure 9.1(a) is a simple but challenging problem for many verification tools. $\mathtt{l}(\mathtt{X},\mathtt{Y}) \equiv \mathtt{X} \geqslant \mathtt{Y} \wedge \mathtt{Y} \geqslant \mathtt{0}$ is a model of the program, whose solution requires the discovery of the invariants $\mathtt{X} \geqslant \mathtt{Y}$ and $\mathtt{Y} \geqslant \mathtt{0}$. For example neither QARMC [65] nor SeaHorn [70] (using only the PDR engine [19]) terminates on this program. However, SeaHorn (with PDR and the abstract interpreter IKOS [20]) solves it. RAHFT solves it with the pre-processing step alone.

RAHFT exploits both of the above formulations using techniques based on *abstract interpretation* over the domain of convex polyhedra, *trace abstraction-refinement* using finite tree automata (FTAs) and *program specialisation* using *constraint specialisation* [94]. The motivations behind this combination are: (i) to benefit from a powerful and scalable technique such as *abstract interpretation* [32] for verifying properties of programs, (ii) to refine *abstract interpretation* through automata theoretic operations which offers the advantages of simplicity and generality [96] and (iii) to construct highly parametric and configurable verification tools through *program transformation* [37].

9.2 ARCHITECTURE AND INTERFACE

Figure 9.1(b) gives an overview of RAHFT. It compiles to a standalone command line utility that accepts a set of CHCs as input. It consists of two modules namely, *Abstraction* (green box) and *Refinement* (red box). RAHFT takes a file containing a set of CHCs P as input and returns *safe* or *unsafe* respectively if P has or does not have a model.

9.2.1 *Abstraction*

The *Abstraction* module takes a set of CHCs P as input and returns *safe*, *unsafe* or a trace representing the abstract derivation of *false* together with the set of all derivations (traces)

(both represented as FTAs) used while applying *abstraction interpretation* to P. It consists of the following components:

Pre-processor (PP): Pre-processing is a model-preserving source-to-source program transformation of Horn clauses. In principle, any such transformation can be used as a pre-processor, but we use *constraint specialisation* [94]. The specialisation consists of strengthening the constraints in the clauses using *abstract interpretation* [32] and *query-answer transformation* [9, 41] of the original program. The specialisation is independent of the *abstract domain* and the background theory underlying the clauses and does not unfold the clauses at all. This has been proven to be an effective transformation [94] for verifying Horn clauses [38] and as a pre-processor to other Horn clause verification tools such as [65].

Abstract Interpreter (AI): The AI implements a fixed point algorithm over the domain of *convex polyhedra* [31] based on *abstract interpretation* [32]. It constructs an over-approximation M of the *minimal model* of a program P, where M contains at most one *constrained fact* $p(X) \leftarrow \phi$ for each predicate p. The constraint ϕ is a conjunction of linear inequalities, representing a convex polyhedron. The set of traces used during abstract interpretation of P can be captured by an FTA, say $\mathcal{A}_P$, using M as shown in [97]. An FTA is a mathematical model capable of capturing tree structured computations (Horn clauses derivations) (see [96] for the correspondence between a program and an FTA).

The approximation M and the pre-processed clauses can be used by other Horn clause tools, for example [65]. These tools can strengthen M (which may contain some useful invariants) incrementally to construct a model of P rather than starting from a coarse abstraction ($p(X) \leftarrow$ *true* for each predicate p of P).

Verifier: The verifier receives M and $\mathcal{A}_P$ and checks the *safety* of the clauses based on the following simple condition. The clauses are safe if there is no constrained fact for *false* in M (M is called *safe inductive invariant* or a *model* of P) or there are no error traces rooted at *false*. Otherwise we do not know whether the clauses are unsafe or whether the approximation was too imprecise. In this case, the verifier picks a trace, say $t \in \mathcal{A}_P$, representing the abstract derivation of *false* (if any) from the set of traces. If t is feasible (while simulating in P), then P is unsafe and t is a *counterexample*, otherwise we refine P.

9.2.2 *Refinement*

The *Refinement* module takes as input a program P and two FTAs (i) recognising the set of all possible traces of P; and (ii) recognising a set of infeasible traces. A difference automaton is computed from these automata which recognises all traces except the infeasible ones. A refined program is obtained as output using the difference automaton and P. Rather than eliminating a single infeasible trace in each refinement iteration, we generalise it using an *interpolant automaton* [74, 97, 147] thereby eliminating a possibly infinite number of infeasible traces. The refinement offers the advantages of simplicity and generality which is independent of the abstract domain and background theory underlying the clauses. The *Refinement* module consists of following components:

Finite tree automata manipulator (FTAM): FTAM takes as input two FTAs and outputs their difference automaton. The FTA difference construction needs *determinisation*; we built upon an

optimised determinisation algorithm by Gallagher, Ajspur and Kafle [61] which scales well in practice, generating transitions of the determinised automaton in a very compact form called *product form*.

Clause generator (CG): Given a set of clauses P, and an automaton recognising an over-approximation of all feasible traces of P, CG produces a set of clauses which is equisatisfiable to P. For this purpose, we exploit a correspondence between the traces using the clauses and the language of FTAs to generate a new set of clauses.

The refinement offers two advantages: (i) the refinement is manifested in the clauses generated – we do not need to keep track of the previous refinements; and (ii) the original predicates get split in refined clauses which help improve the precision of analysis [59].

9.2.3 *Implementation*

RAHFT is implemented in Ciao [78] and is available from `https://github.com/bishoksan/RAHFT`. It consists of a collection of reusable Prolog modules which rely on state-of-the-art specialised external libraries written in C and C++ for handling constraints. We use the Yices SMT solver [44] and the Parma Polyhedra Library [5] for handling the constraints and the FTA library [61] for manipulating FTAs. The construction of an *interpolant tree automaton* uses an algorithm presented in [138] for computing an interpolant of two formulas. The code consists of over 7,000 lines of Ciao Prolog code split over 42 modules, interfaced to the above-mentioned external libraries. The implementation of iterative fixpoint algorithms is inspired by the approach to the abstract interpretation of logic programs described by Codish and Søndergaard [27]. Data structures for manipulating Horn clauses are based on terms and the internal Prolog database, reusing the optimizations of the underlying machine (e.g., clause indexing) rather than reimplementing them in our tool. The glue code that ties together the general purpose Prolog engine and the specialised solvers written in C and C++ is generated via the Ciao foreign interface [78].

9.2.4 *Strength and weakness*

RAHFT is a verification tool for *safety* properties of programs expressed as Horn clauses; it can be used as a *back end* solver by different *front end* tools outputting in CLP form. It handles clauses whose underlying theory is *linear arithmetic*; other theories are not supported currently. It accepts input in CLP form.

Since different components of RAHFT are loosely coupled, the tool can be reconfigured (with a very little effort) to produce verification tools solely based on (i) program transformation as in iterated specialisation approach [38] by iterating the pre-processing component, (ii) abstract interpretation, only with the AI component, (iii) trace abstraction refinement [74, 147] by iterating the FTAM component, and (iv) a sensible combination thereof – all followed by a lightweight verifier which checks the *safety* of the clauses based on the conditions mentioned

above. Since our tool uses both *state abstraction* and *trace abstraction*, it allows application of a wide range of tools and techniques.

We have evaluated RAHFT on software verification benchmarks from a variety of sources [67, 85, 68, 12, 13, 80] and the results show that it compares favourably (in time and the number of instances solved) with the other state-of-the-art Horn clause verification tools (see [96, 94, 97] for the details).

Convex polyhedra is an expensive abstract domain and is a potential bottleneck for verification of large code bases. Instead, we can use cheaper domains supported by the Parma Polyhedra Library such as *octagons* or *intervals* at the cost of precision. RAHFT is also limited by the *hard-coded* limits of the libraries and the Prolog implementation used (e.g. *arity* limit of the predicates), which may be too restrictive for some verification problems and we intend to improve this by some suitable data representation. We are aware of some examples from SV-COMP if not many which cross this limit.

We can leverage state-of-the-art interpolating SMT solvers [22, 125] for the *tree interpolant* generation which can be used for constructing an *interpolant tree automaton*; our current implementation does not scale well. Furthermore we aim to handle more advanced data structures such as arrays, maps and sets, requiring more expressive theories than linear arithmetic. One way to achieve this is by composing abstract domains as described in [34, 29]; we are also aware of the support for the reduced product of domains in the PPL library.

RAHFT is able to generate a *model* (*a counterexample*) if it proves the *safety* (*unsafety*) a program. We need bookkeeping to generate these witnesses with respect to the original program; and sometimes it becomes rather challenging because of the use of external libraries, tools or the transformations applied.

9.3 FUTURE WORK

Future work will involve making RAHFT a more flexible tool so that the user can configure other parameters such as abstract domains and pre-processors. We are also planning for a detailed performance measurement of the tool to detect bottlenecks; and work on language-based optimisations to minimize them. Generation of a *model* or a *counterexample* with respect to the original program, handling clauses with richer background theories (arrays, uninterpreted functions) is on our to-do list. In addition, we are extending RAHFT to consider Horn clauses in `SMT-LIB` format [1], though several Horn clause verification tools use standard CLP notation [96, 37, 65].

10 CONCLUSION AND FUTURE WORK

This thesis described an approach for verifying sets of Horn clauses; these usually represent safety properties of (imperative) programs. The approach presented, extended and combined program transformation, abstract interpretation, and trace abstraction refinement into a single tool for Horn clause verification. The combination led to a component-based *abstraction refinement* framework for Horn clause verification. Next we summarise the achievements of the research described in the thesis and its possible extensions.

10.1 PROGRAM TRANSFORMATION FOR HORN CLAUSE VERIFICATION

SUMMARY OF ACHIEVEMENTS. The program transformations we propose for Horn clauses have the goals of propagating constraints throughout the program, splitting program predicates, removing redundant variables from the program and removing some program parts irrelevant to the properties in question – which have some surprisingly beneficial effects on verification. These transformations were used as pre-processors to Horn clauses verification tools and sometimes as proof techniques for Horn clauses. In Chapter 3 and 4, we described several transformations for Horn clauses and their role in Horn clause verification.

The transformations that were especially important in our experiments are predicate splitting and constraint specialisation. Firstly, predicate splitting based on a given criterion creates separate version of a predicate each defined by its own clauses. The criterion we used was to split a predicate defined by the clauses with mutually exclusive constraints. Disjunctive invariants are needed to prove many program properties and disjunctive solvers are computationally expensive. This transformation allows discovering disjunctive properties using a conjunctive solver. Secondly, constraint specialisation, a method of program transformation specialises the constraints in the clauses with respect to a property to be verified (goal). It simultaneously propagates the constraints from the goal and from the constrained facts. The surprising fact was that many programs were proven just by the transformation alone. Furthermore the iteration of the method caused more problems to be solved. Not only this, these transformations can also be used as a pre-processor for other Horn clause verification tools – improving their effectiveness.

FUTURE WORK. Predicate splitting seems to be crucial in Horn clause verification. It allows simulating disjunctions which are needed to prove many properties. Different heuristics for splitting produces different sets of clauses which may have different impact in verification. We would like to explore and experiment with predicate splitting heuristics evaluating their effects on verification.

Since the method of constraint specialisation exposes implicit constraints from clauses, this information can be exploited in several program analysis tasks. One such task is program

debugging. A specialised program has more explicit information which is available to all debugging tools, which may make errors in the original program apparent. If the constraints in a clause strengthen to *false*, then this explains why this clause cannot be used in any derivation to derive a goal. Another task is iterated specialisation. The explicit information may allow further specialisations for example by providing some knowledge of the call context of each program point. Further it can have application in termination analysis. Discovery of a ranking function is central to termination analysis. A ranking function is some expression over the program variables, which evaluates to a non-negative value and strictly decreases in each iteration of the loop. The explicit constraints open the possibility of discovering tighter ranking functions.

10.2 ABSTRACT INTERPRETATION FOR FINDING INVARIANTS

SUMMARY OF ACHIEVEMENTS. Proving properties of programs reduces to finding invariants of programs. So we analysed a set of Horn clauses using abstract interpretation over the domain of convex polyhedra, an abstract domain capable of representing useful numeric (linear arithmetic) invariants necessary to prove many properties. The fixed point algorithm over this domain as presented in Chapter 4 proved to be a very powerful verification tool in itself. This has been demonstrated on a set of Horn clause verification problems. The operations (convex hull and widening) over this domain are computationally expensive and depend on the dimensions of polyhedra (the number of variables). We performed live variable analysis (described as *redundant argument filtering* in Chapter 3) of the program to reduce the number of variables whenever possible before applying abstract interpretation, which has a positive effect in the verification problems. Furthermore we presented a method for generating threshold constraints from the Horn clause program which can be used during widening (as widening with thresholds) operation to control the loss the precision due to widening.

FUTURE WORK. There are at least two promising avenues for future work here. On the one hand, the domain of convex polyhedra is expensive and is a potential bottleneck for verification of large programs. Instead, we can use cheaper domains such as *octagons* or *intervals* at the cost of precision. So we are currently extending our work to incorporate these abstract domains. On the other hand, our attempt so far has been on verifying Horn clauses expressed over the theory of linear arithmetic. Usually programs contain more advanced data structures such as arrays, maps and sets which require more expressive theories than linear arithmetic to handle them. We plan to analyse these programs over the richer background theories by composing abstract domains (for example the reduced product of abstract domains) or by combining decision procedures for the richer theories as described in [34, 29].

10.3 TRACE ABSTRACTION REFINEMENT

SUMMARY OF ACHIEVEMENTS. Abstract interpretation is a scalable technique but often suffers from *false alarms*. Techniques have been proposed in the literature which refine ab-

stractions, for example predicate abstraction but much less efforts have been spend on refining abstract interpretations. In addition, the domain of convex polyhedra we used is not directly amenable to refinement since it is fixed. Instead, we proposed tree-automata techniques to refine traces derived from an abstract interpretation of a set of Horn clauses in Chapter 5 which induces refinement of programs. Indirectly this can be viewed as refinement of abstraction interpretations through the refinement of programs.

Tree-automata techniques offer a method for manipulating sets of tree-structured traces. An overapproximation of a set of all traces of Horn clauses and a set of spurious traces for the predicate specifying an error state are represented by FTAs. The FTA operation removes these spurious traces from the set of all traces using the difference automaton construction. This can be viewed as trace refinement. Furthermore, we use the concept of *interpolant tree automata* to discover more infeasible traces by trace generalisation and remove them in a single refinement step. Then a new program is generated from these refined set of traces. There are number of achievements reported in Chapter 5. Firstly, we showed how abstract interpretation interacts with FTAs. Secondly, we exploited the correspondence between FTAs and Horn clause derivations. This allows transformations in Horn clause to be achieved through transformations of FTAs. The refinement offers a number of advantages. First, the refinement is manifested in the clauses generated – we do not need to keep track of the previous refinements. Second, the original predicates get split in refined clauses which helps to improve the precision of the analysis. Third, the refinement offers the advantages of simplicity and generality which is independent of the abstract domain and background theory underlying the clauses. Finally, the practicality of our approach was demonstrated on a set of Horn clause verification problems.

FUTURE WORK. We will investigate different heuristics for interpolation generation which may give rise to different interpolant automaton and it may have different impact on verification. At the moment, a new program is generated after each refinement and the analysis is restarted from scratch. In the future, we would like to reuse the result of analysis from the previous iterations and build on this instead of starting the analysis from scratch. The results from previous analyses could be the abstraction (narrowing techniques can be applied from this abstraction) or the interpolants (which can be used as threshold constraints). Further study is needed to find a suitable combination of abstract interpretation and interpolation based techniques, based on a deeper understanding of the interaction among interpolation, trace elimination and abstract interpretation.

10.4 DECOMPOSITION OF THE VERIFICATION PROBLEM

SUMMARY OF ACHIEVEMENTS. Proof decomposition allows complex proofs to be split into simpler ones. We proposed a proof decomposition technique for Horn clauses, based on the concept of *tree dimension* of Horn clause derivation (a measure of its non-linearity) in Chapter 7. A proof of a set of CHCs can be decomposed into several proofs for different values of *tree dimension*, which can be computed in parallel. In this way, the proof for the original set of CHCs can be composed from the proofs of its constituents.

FUTURE WORK. The idea we pursued seems interesting because the technique allows to decompose programs into several sub-programs based on different values of *tree dimension* in a syntactic way. Then the proofs for each such sub-programs can be combined. However, we did not get much success with our decomposition technique because a program can have an unbounded *tree dimension* and the proof of sub-programs of a certain dimension can be as complex as the original one. There are at least two lines of research on program verification based on tree-dimension that are worth investigating. They are proof by induction on dimension and bounded model checking [14] based on dimension (dimension-bounded).

10.5 SUFFICIENCY OF A LINEAR SOLVER

SUMMARY OF ACHIEVEMENTS. We presented an *abstraction-refinement* approach for solving a set of non-linear Horn clauses using available linear Horn clause solvers in Chapter 8. The approach is based on a linearisation of a dimension-bounded set of Horn clauses (which are known to be linearisable) using partial evaluation and the use of a linear Horn clause solver. Experiment on a set of non-linear Horn clause verification problems using our approach showed that the approach is feasible (a linear solver can be used for solving non-linear problems) and the solvability of a problem is often shallow with respect to its dimension.

FUTURE WORK. Further study is needed to understand whether there are benefits of linear Horn clause solving over the non-linear ones. We would like to get a deeper understanding of the solving algorithms and techniques used for each class of programs (linear and non-linear) and compare them. The solvability of a problem depends on the linearisation strategies used, for example depth first or breadth first. So we also plan to experiment with different linearisation strategies for Horn clauses and study their effects in verification. Linearisation of Horn clauses is not possible in general, so we would like to find a linearisable class of Horn clauses potentially useful in practice.

10.6 TOOLS

SUMMARY OF ACHIEVEMENTS. The ideas presented in this thesis are implemented in the tools *LHornSolver*[1] and RAHFT[2] and the tools' descriptions are given in Chapters 8 and 9 respectively. Both of these tools follow an abstraction refinement approach for solving a set of Horn clauses. The first one does so non-linearly, the second linearly, with the same underlying solver for Horn clauses. They take as input a set of Horn clauses in CLP syntax and return *safe* or *unsafe*. The goal of *LHornSolver* is to allow linear Horn clause solvers to be applied to solve non-linear Horn clauses. The experimental results suggest that it is a feasible approach but we did not find any case where the use of a linear solver could be an advantage over the non-linear ones. This could be due to the linearisation strategies used or the use of a solver not optimised for solving linear programs. We have evaluated RAHFT on software

1 https://github.com/bishoksan/LHornSolver
2 https://github.com/bishoksan/RAHFT

verification benchmarks and the results showed that it compares favourably (in time and the number of instances solved) with the other state-of-the-art Horn clause verification tools. There are problems which are not solvable by any state of the art tools, solvable by very few tools including RAHFT and solvable by some of the tools but not by RAHFT. The techniques employed by the tools and their foci differ significantly. Some tools focus on finding bugs (tools based on bounded model checking) whereas some focus on proving programs (tools based on abstract interpretation). Further, the tools in the literature exploit different program structures (linear or non-linear). Tools also differ in the way they compute invariants to prove a program. Some derive invariants using a fixed point algorithm over some abstract domain using abstract interpretation; some do so by generalising from a specific counterexample possibly using interpolation and some do so by using generalisation operators (this is case of transformational approach to verification such as VeriMap). RAHFT computes invariants using abstract interpretation over the domain of convex polyhedra and is usually good for proving programs. The approach for Horn clause verification proposed in this thesis allows several different techniques and tools developed for Horn clauses to be combined in a flexible way and experiment with them. The tool RAHFT is a snapshot of a particular combination, not necessarily an optimal one. The component-based approach allows components to be reconfigured with a very little effort saving time and effort. RAHFT can also serve as a back end solver for different front end tools outputting in CLP form.

FUTURE WORK. Currently we are extending the tools into several directions. First of all, we want to consider Horn clauses in `SMT-LIB` format [1] (the standard format for SMT solvers) to be able to accept a wider range of input, though several Horn clause verification tools use standard CLP notation [96, 37, 65]. Second of all, we want to make these tools more flexible so that the user can configure other parameters such as abstract domains and pre-processors. The optimisation of our tools is an important topic for future work. The generation of witnesses (*model* or a *counterexample*) with respect to the original program is another area for investigation. Handling clauses with richer background theories (arrays, uninterpreted functions) are on our to-do list. Finding a best combination of techniques and tools which allows us to solve as many problems as possible is an ongoing task.

Horn clauses are suitable representation not only for verification but also for other program analysis tasks. Many program analysis tasks, for example program equivalence [40] can be reduced to Horn clauses solving. We are planning to extend the tools and techniques developed in this thesis beyond safety verification, for example resource analysis, termination analysis and liveness analysis.

A

BACKGROUND READING

We would like this thesis to be readable by as many people as possible, but it does presuppose background knowledge in several areas. So, for interested readers who would like to know more, we suggest the following.

A.1 INTRODUCTORY MATERIAL

- Lloyd, J. Foundations of Logic Programming: 2nd Edition. Springer-Verlag, 1987.
- Huth, M. and Ryan, M. D. Logic in computer science - modelling and reasoning about systems, Second Edition. Cambridge University Press, 2004.
- Fitting, M. First-Order Logic and Automated Theorem Proving, Second Edition. Graduate Texts in Computer Science. Springer, 1996.
- Pettorossi A. and Proietti M. First order predicate calculus and logic programming, Third Edition. ARACNE, 2014.

A.2 ADVANCED MATERIAL

- Jaffar, J. and Maher, M. Constraint Logic Programming: A Survey. Journal of Logic Programming, 19/20:503-581, 1994.
- Jhala, R. and Majumdar, R. Software model checking. ACM Comput. Surv., 41(4), 2009.
- Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors. Handbook of Satisfiability, volume 185 of Frontiers in Artificial Intelligence and Applications, 2009. IOS Press.
- Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

A.3 KEY RESEARCH PAPERS

- Cousot, P. and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Graham, R. M., Harrison, M. A., and Sethi, R., editors, POPL, pages 238-252. ACM, 1977.
- Cousot, P. and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In Aho, A. V., Zilles, S. N., and Szymanski, T. G., editors, Conference

Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978, pages 84-96. ACM Press, 1978.

- Bagnara, R., Hill, P. M., and Zaffanella, E. The Parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Sci. Comput. Program., 72(1-2):3-21, 2008.
- Benoy, F. and King, A. Inferring argument size relationships with CLP(R). In Gallagher, J. P., editor, Logic-Based Program Synthesis and Transformation (LOPSTR'96), volume 1207 of LNCS, pages 204-223, August 1996.
- Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. J. ACM, 50(5):752-794, 2003.
- Codish, M. Efficient goal directed bottom-up evaluation of logic programs. J. Log. Program., 38(3):355-370, 1999.
- De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. Program verification via iterated specialization. Sci. Comput. Program., 95:149-175, 2014.
- Gallagher, J. P. and Lafave, L. Regular approximation of computation paths in logic and functional languages. In Danvy, O., Glück, R., and Thiemann, P., editors, Partial Evaluation, volume 1110 of Springer-Verlag LNCS, pages 115-136, 1996.
- Graf, S. and Saïdi, H. Construction of abstract state graphs with PVS. In Grumberg, O., editor, Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings, volume 1254 of LNCS, pages 72-83. Springer, 1997.
- Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. Synthesizing software verifiers from proof rules. In Vitek, J., Lin, H., and Tip, F., editors, ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012, pages 405-416. ACM, 2012.
- Heizmann, M., Hoenicke, J., and Podelski, A. Refinement of trace abstraction. In Palsberg, J. and Su, Z., editors, Static Analysis, 16th International Symposium, SAS 2009, volume 5673 of LNCS, pages 69-85. Springer, 2009.
- Jhala, R. and McMillan, K. L. A practical and complete approach to predicate refinement. In Hermanns, H. and Palsberg, J., editors, TACAS, volume 3920 of LNCS, pages 459-473. Springer, 2006.
- Leuschel, M. Advanced logic program specialisation. In Hatcliff, J., Mogensen, T. A., and Thiemann, P., editors, Partial Evaluation - Practice and Theory, volume 1706 of LNCS, pages 271-292. Springer, 1999.
- Levi, G. Abstract interpretation based verification of logic programs. Electr. Notes Theor. Comput. Sci., 40:243, 2000.

• Peralta, J., Gallagher, J. P., and Saglam, H. Analysis of imperative programs through analysis of constraint logic programs. In Levi, G., editor, Static Analysis. 5th International Symposium, SAS'98, Pisa, volume 1503 of Springer-Verlag LNCS, pages 246-261, 1998.

BIBLIOGRAPHY

[1] SMT-LIB format. Available on: http://smtlib.cs.uiowa.edu. Accessed: 2016-05-05.

[2] Abelson, H. and Sussman, G. J. *Structure and Interpretation of Computer Programs, Second Edition*. MIT Press, 1996.

[3] Afrati, F. N., Gergatsoulis, M., and Toni, F. Linearisability on datalog programs. *Theor. Comput. Sci.*, 308(1-3):199–226, 2003.

[4] Albarghouthi, A., Gurfinkel, A., and Chechik, M. Craig interpretation. In Miné, A. and Schmidt, D., editors, *SAS*, volume 7460 of *LNCS*, pages 300–316. Springer, 2012.

[5] Bagnara, R., Hill, P. M., and Zaffanella, E. The Parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. *Sci. Comput. Program.*, 72(1-2):3–21, 2008.

[6] Baier, C. and Tinelli, C., editors. *TACAS. Proceedings*, volume 9035 of *LNCS*, 2015. Springer.

[7] Ball, T., Podelski, A., and Rajamani, S. K. Relative completeness of abstraction refinement for software model checking. In Katoen, J. and Stevens, P., editors, *Tools and Algorithms for the Construction and Analysis of Systems, 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2280 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 2002.

[8] Ball, T., Levin, V., and Rajamani, S. K. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[9] Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. Magic sets and other strange ways to implement logic programs. In *Proceedings of the 5th ACM SIGMOD-SIGACT Symposium on Principles of Database Systems*, 1986.

[10] Benoy, F. and King, A. Inferring argument size relationships with CLP(R). In Gallagher, J. P., editor, *Logic-Based Program Synthesis and Transformation (LOPSTR'96)*, volume 1207 of *LNCS*, pages 204–223, August 1996.

[11] Beyer, D. Repository of Horn clauses. https://github.com/sosy-lab/sv-benchmarks/tree/master/clauses, Nov. 2012. Accessed: 2016-04-18.

[12] Beyer, D. Second competition on software verification - (summary of SV-COMP 2013). In Piterman and Smolka [133], pages 594–609.

[13] Beyer, D. Software verification and verifiable witnesses - (report on SV-COMP 2015). In Baier and Tinelli [6], pages 401–416.

[14] Biere, A. Bounded model checking. In Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 457–481. IOS Press, 2009.

[15] Bjørner, N., McMillan, K. L., and Rybalchenko, A. On solving universally quantified Horn clauses. In Logozzo, F. and Fähndrich, M., editors, *SAS*, volume 7935 of *LNCS*, pages 105–125. Springer, 2013.

[16] Bjørner, N., Fioravanti, F., Rybalchenko, A., and Senni, V., editors. *Proceedings First Workshop on Horn Clauses for Verification and Synthesis, HCVS 2014, Vienna, Austria, 17 July 2014*, volume 169 of *EPTCS*, 2014.

[17] Bjørner, N., Gurfinkel, A., McMillan, K. L., and Rybalchenko, A. Horn clause solvers for program verification. In Beklemishev, L. D., Blass, A., Dershowitz, N., Finkbeiner, B., and Schulte, W., editors, *Fields of Logic and Computation II - Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*, volume 9300 of *LNCS*, pages 24–51. Springer, 2015.

[18] Blanc, R., Gupta, A., Kovács, L., and Kragl, B. Tree interpolation in vampire. In McMillan, K. L., Middeldorp, A., and Voronkov, A., editors, *LPAR*, volume 8312 of *LNCS*, pages 173–181. Springer, 2013.

[19] Bradley, A. R. and Manna, Z. Property-directed incremental invariant generation. *Formal Asp. Comput.*, 20(4-5):379–405, 2008.

[20] Brat, G., Navas, J. A., Shi, N., and Venet, A. IKOS: A framework for static analysis based on abstract interpretation. In Giannakopoulou, D. and Salaün, G., editors, *SEFM*, volume 8702 of *LNCS*, pages 271–277. Springer, 2014.

[21] Burke, M. and Soffa, M. L., editors. *Proceedings of the 2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Snowbird, Utah, USA, June 20-22, 2001*, 2001. ACM.

[22] Cimatti, A., Griggio, A., Schaafsma, B. J., and Sebastiani, R. The MathSAT5 SMT solver. In Piterman and Smolka [133], pages 93–107.

[23] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003.

[24] Codish, M. and Demoen, B. Analysing logic programs using 'prop'-ositional logic programs and a magic wand. In Miller, D., editor, *Proceedings of the 1993 International Symposium on Logic Programming, Vancouver*. MIT Press, 1993.

[25] Codish, M. Efficient goal directed bottom-up evaluation of logic programs. *J. Log. Program.*, 38(3):355–370, 1999.

[26] Codish, M. and Demoen, B. Analyzing logic programs using "PROP"-ositional logic programs and a magic wand. *J. Log. Program.*, 25(3):249–274, 1995.

[27] Codish, M. and Søndergaard, H. Meta-circular abstract interpretation in Prolog. In Mogensen, T. Æ., Schmidt, D. A., and Sudborough, I. H., editors, *The Essence of Computation, Complexity, Analysis, Transformation*, volume 2566 of *LNCS*, pages 109–134. Springer, 2002.

[28] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. Tree automata techniques and applications. Available on: http://www.grappa.univ-lille3.fr/tata, 2007. release October, 12th 2007.

[29] Cortesi, A., Costantini, G., and Ferrara, P. A survey on product operators in abstract interpretation. In Banerjee, A., Danvy, O., Doh, K., and Hatcliff, J., editors, *Semantics, Abstract Interpretation, and Reasoning about Programs*, volume 129 of *EPTCS*, pages 325–336, 2013.

[30] Cousot, P. and Cousot, R. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[31] Cousot, P. and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In *Proceedings of the 5th Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, 1978.

[32] Cousot, P. and Cousot, R. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Graham, R. M., Harrison, M. A., and Sethi, R., editors, *POPL*, pages 238–252. ACM, 1977.

[33] Cousot, P. and Halbwachs, N. Automatic discovery of linear restraints among variables of a program. In Aho, A. V., Zilles, S. N., and Szymanski, T. G., editors, *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, USA, January 1978*, pages 84–96. ACM Press, 1978.

[34] Cousot, P., Cousot, R., and Mauborgne, L. The reduced product of abstract domains and the combination of decision procedures. In Hofmann, M., editor, *FOSSACS 2011*, volume 6604 of *LNCS*, pages 456–472. Springer, 2011.

[35] Craig, W. Linear reasoning. A new form of the Herbrand-Gentzen theorem. *J. Symb. Log.*, 22(3):250–268, 1957.

[36] Dawson, S., Ramakrishnan, C. R., Ramakrishnan, I. V., and Sekar, R. C. Extracting determinacy in logic programs. In Warren, D. S., editor, *Logic Programming, Proceedings of the Tenth International Conference on Logic Programming, Budapest, Hungary, June 21-25, 1993*, pages 424–438. MIT Press, 1993.

[37] De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. Verimap: A tool for verifying programs through transformations. In Ábrahám, E. and Havelund, K., editors, *TACAS*, volume 8413 of *LNCS*, pages 568–574. Springer, 2014.

[38] De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. Program verification via iterated specialization. *Sci. Comput. Program.*, 95:149–175, 2014.

[39] De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. Semantics-based generation of verification conditions by program specialization. In Falaschi, M. and Albert, E., editors, *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pages 91–102. ACM, 2015.

[40] De Angelis, E., Fioravanti, F., Pettorossi, A., and Proietti, M. Proving correctness of imperative programs by linearizing constrained Horn clauses. *TPLP*, 15(4-5):635–650, 2015.

[41] Debray, S. and Ramakrishnan, R. Abstract Interpretation of Logic Programs Using Magic Transformations. *Journal of Logic Programming*, 18:149–176, 1994.

[42] Debray, S. K. and Ramakrishnan, R. Abstract interpretation of logic programs using magic transformations. *J. Log. Program.*, 18(2):149–176, 1994.

[43] Delzanno, G. and Podelski, A. Constraint-based deductive model checking. *STTT*, 3(3): 250–270, 2001.

[44] Dutertre, B. Yices 2.2. In Biere, A. and Bloem, R., editors, *Computer-Aided Verification (CAV'2014)*, volume 8559 of *LNCS*, pages 737–744. Springer, July 2014.

[45] Esparza, J., Kiefer, S., and Luttenberger, M. On fixed point equations over commutative semirings. In *STACS, Proceedings*, volume 4393 of *LNCS*, pages 296–307. Springer, 2007.

[46] Esparza, J., Kiefer, S., and Luttenberger, M. Newtonian program analysis. *J. ACM*, 57 (6):33, 2010.

[47] Esparza, J., Ganty, P., Kiefer, S., and Luttenberger, M. Parikh's theorem: A simple and direct automaton construction. *Inf. Process. Lett.*, 111(12):614–619, 2011.

[48] Esparza, J., Luttenberger, M., and Schlund, M. A brief history of strahler numbers. In Dediu, A. H., Martín-Vide, C., Sierra-Rodríguez, J. L., and Truthe, B., editors, *LATA. Proceedings*, volume 8370 of *LNCS*, pages 1–13. Springer, 2014.

[49] Fitting, M. *First-Order Logic and Automated Theorem Proving, Second Edition*. Graduate Texts in Computer Science. Springer, 1996.

[50] Fujita, H. An algorithm for partial evaluation with constraints. Technical Report TR-258, ICOT, 1987.

[51] Gallagher, J. P. Transforming logic programs by specialising interpreters. In *Proceedings of the 7th European Conference on Artificial Intelligence (ECAI-86), Brighton*, pages 109–122, 1986.

[52] Gallagher, J. P. Specialisation of logic programs: A tutorial. In *Proceedings PEPM'93, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 88–98, Copenhagen, June 1993. ACM Press.

[53] Gallagher, J. P. and Bruynooghe, M. Some low-level source transformations for logic programs. In *Proceedings of Meta90 Workshop on Meta Programming in Logic*. Katholieke Universiteit Leuven, Belgium, 1990.

[54] Gallagher, J. P. and de Waal, D. Deletion of redundant unary type predicates from logic programs. In Lau, K. and Clement, T., editors, *Logic Program Synthesis and Transformation*, Workshops in Computing, pages 151–167. Springer-Verlag, 1993.

[55] Gallagher, J. P. and de Waal, D. Fast and precise regular approximation of logic programs. In Van Hentenryck, P., editor, *Proceedings of the International Conference on Logic Programming (ICLP'94), Santa Margherita Ligure, Italy*. MIT Press, 1994.

[56] Gallagher, J. P. and Lafave, L. Regular approximation of computation paths in logic and functional languages. In Danvy, O., Glück, R., and Thiemann, P., editors, *Partial Evaluation*, volume 1110 of *Springer-Verlag LNCS*, pages 115–136, 1996.

[57] Gallagher, J. P., Codish, M., and Shapiro, E. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6:159–186, 1988.

[58] Gallagher, J. P. Tutorial on specialisation of logic programs. In Schmidt, D. A., editor, *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM)*, pages 88–98. ACM, 1993.

[59] Gallagher, J. P. and Kafle, B. Analysis and transformation tools for constrained Horn clause verification. *TPLP*, 14(4-5 (additional materials in online edition)):90–101, 2014.

[60] Gallagher, J. P., Ajspur, M., and Kafle, B. An Optimised Algorithm for Determinisation and Completion of Finite Tree Automata. Technical Report 145, Roskilde University, Denmark, 09 2014. Available from `http://akira.ruc.dk/~jpg/dfta.pdf`.

[61] Gallagher, J. P., Ajspur, M., and Kafle, B. An optimised algorithm for determinisation and completion of finite tree automata. *CoRR*, abs/1511.03595, 2015.

[62] Gange, G., Navas, J. A., Schachte, P., Søndergaard, H., and Stuckey, P. J. Failure tabled constraint logic programming by interpolation. *TPLP*, 13(4-5):593–607, 2013.

[63] Ganty, P., Iosif, R., and Konecný, F. Underapproximation of procedure summaries for integer programs. In Piterman and Smolka [133], pages 245–259.

[64] Grebenshchikov, S., Gupta, A., Lopes, N. P., Popeea, C., and Rybalchenko, A. HSF(C): A software verifier based on Horn clauses - (competition contribution). In Flanagan, C. and König, B., editors, *TACAS*, volume 7214 of *LNCS*, pages 549–551. Springer, 2012.

[65] Grebenshchikov, S., Lopes, N. P., Popeea, C., and Rybalchenko, A. Synthesizing software verifiers from proof rules. In Vitek, J., Lin, H., and Tip, F., editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 405–416. ACM, 2012.

[66] Gruska, J. A few remarks on the index of context-free grammars and languages. *Information and Control*, 19(3):216–223, 1971.

[67] Gulavani, B. S., Chakraborty, S., Nori, A. V., and Rajamani, S. K. Automatically refining abstract interpretations. In Ramakrishnan, C. R. and Rehof, J., editors, *TACAS*, volume 4963 of *LNCS*, pages 443–458. Springer, 2008.

[68] Gupta, A. and Rybalchenko, A. Invgen: An efficient invariant generator. In Bouajjani, A. and Maler, O., editors, *CAV*, volume 5643 of *LNCS*, pages 634–640. Springer, 2009.

[69] Gupta, A., Popeea, C., and Rybalchenko, A. Solving recursion-free Horn clauses over LI+UIF. In Yang, H., editor, *Programming Languages and Systems - 9th Asian Symposium, APLAS 2011, Kenting, Taiwan, December 5-7, 2011. Proceedings*, volume 7078 of *LNCS*, pages 188–203. Springer, 2011.

[70] Gurfinkel, A., Kahsai, T., Komuravelli, A., and Navas, J. A. The seaHorn verification framework. In Kroening, D. and Pasareanu, C. S., editors, *CAV , Proceedings, Part I*, volume 9206 of *LNCS*, pages 343–361. Springer, 2015.

[71] Gurfinkel, A., Kahsai, T., and Navas, J. A. SeaHorn: A framework for verifying C programs (competition contribution). In Baier and Tinelli [6], pages 447–450.

[72] Halbwachs, N., Proy, Y. E., and Raymound, P. Verification of linear hybrid systems by means of convex approximations. In *Proceedings of the First Symposium on Static Analysis*, volume 864 of *LNCS*, pages 223–237, September 1994.

[73] Halbwachs, N., Proy, Y. E., and Raymound, P. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[74] Heizmann, M., Hoenicke, J., and Podelski, A. Refinement of trace abstraction. In Palsberg, J. and Su, Z., editors, *Static Analysis, 16th International Symposium, SAS 2009*, volume 5673 of *LNCS*, pages 69–85. Springer, 2009.

[75] Heizmann, M., Hoenicke, J., and Podelski, A. Nested interpolants. In Hermenegildo, M. V. and Palsberg, J., editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 471–482. ACM, 2010.

[76] Heizmann, M., Hoenicke, J., and Podelski, A. Nested interpolants. In Hermenegildo, M. V. and Palsberg, J., editors, *Proceedings of POPL 2010*, pages 471–482. ACM, 2010.

[77] Heizmann, M., Hoenicke, J., and Podelski, A. Software model checking for people who love automata. In Sharygina and Veith [140], pages 36–52.

[78] Hermenegildo, M. V., Bueno, F., Carro, M., López-García, P., Mera, E., Morales, J. F., and Puebla, G. An overview of ciao and its design philosophy. *TPLP*, 12(1-2):219–252, 2012.

[79] Hoder, K. and Bjørner, N. Generalized property directed reachability. In Cimatti, A. and Sebastiani, R., editors, *SAT. Proceedings*, volume 7317 of *LNCS*, pages 157–171. Springer, 2012.

[80] Hojjat, H., Konecný, F., Garnier, F., Iosif, R., Kuncak, V., and Rümmer, P. A verification toolkit for numerical transition systems - tool paper. In Giannakopoulou, D. and Méry, D., editors, *FM 2012: Formal Methods - 18th International Symposium, Paris, France, August 27-31, 2012. Proceedings*, volume 7436 of *LNCS*, pages 247–251. Springer, 2012.

[81] Jaffar, J. and Maher, M. J. Constraint logic programming: A survey. *J. Log. Program.*, 19/20:503–581, 1994.

[82] Jaffar, J., Maher, M. J., Marriott, K., and Stuckey, P. J. The semantics of constraint logic programs. *J. Log. Program.*, 37(1-3):1–46, 1998.

[83] Jaffar, J., Santosa, A. E., and Voicu, R. Modeling systems in CLP. In Gabbrielli, M. and Gupta, G., editors, *Logic Programming, 21st International Conference, ICLP 2005, Sitges, Spain, October 2-5, 2005, Proceedings*, volume 3668 of *LNCS*, pages 412–413. Springer, 2005.

[84] Jaffar, J., Navas, J. A., and Santosa, A. E. Unbounded symbolic execution for program verification. In Khurshid, S. and Sen, K., editors, *RV*, volume 7186 of *LNCS*, pages 396–411. Springer, 2011.

[85] Jaffar, J., Murali, V., Navas, J. A., and Santosa, A. E. Tracer: A symbolic execution tool for verification. In Madhusudan, P. and Seshia, S. A., editors, *CAV*, volume 7358 of *LNCS*, pages 758–766. Springer, 2012.

[86] Jhala, R. and Majumdar, R. Software model checking. *ACM Comput. Surv.*, 41(4), 2009.

[87] Jhala, R. and McMillan, K. L. A practical and complete approach to predicate refinement. In Hermanns, H. and Palsberg, J., editors, *TACAS*, volume 3920 of *LNCS*, pages 459–473. Springer, 2006.

[88] Jones, N. and Mycroft, A. Dataflow analysis of applicative programs using minimal function graphs. In *Proceedings of Principle of Programming Languages (POPL'86)*. ACM Press, 1986.

[89] Jones, N., Gomard, C., and Sestoft, P. *Partial Evaluation and Automatic Software Generation*. Prentice Hall, 1993.

[90] Jones, N. D. Combining abstract interpretation and partial evaluation. In Van Hentenryck, P., editor, *Symposium on Static Analysis (SAS'97)*, volume 1302 of *Springer-Verlag LNCS*, pages 396–405, 1997.

[91] Jones, N. D. Transformation by interpreter specialisation. *Sci. Comput. Program.*, 52: 307–339, 2004.

[92] Jones, N. D. and Rosendahl, M. Higher-order minimal function graphs. *Journal of Functional and Logic Programming*, 1997(2), 1997.

[93] Kafle, B. and Gallagher, J. P. Convex polyhedral abstractions, specialisation and property-based predicate splitting in horn clause verification. In Bjørner et al. [16], pages 53–67.

[94] Kafle, B. and Gallagher, J. P. Constraint specialisation in Horn clause verification. In Asai, K. and Sagonas, K., editors, *Proceedings of the 2015 Workshop on Partial Evaluation and Program Manipulation, PEPM, Mumbai, India, January 15-17, 2015*, pages 85–90. ACM, 2015.

[95] Kafle, B. and Gallagher, J. P. Tree automata-based refinement with application to Horn clause verification. In D'Souza, D., Lal, A., and Larsen, K. G., editors, *VMCAI. Proceedings*, volume 8931 of *LNCS*, pages 209–226. Springer, 2015.

[96] Kafle, B. and Gallagher, J. P. Horn clause verification with convex polyhedral abstraction and tree automata-based refinement. *Computer Languages, Systems & Structures*, 2015.

[97] Kafle, B. and Gallagher, J. P. Interpolant tree automata and their application in Horn clause verification. In Hamilton, G. W., Lisitsa, A., and Nemytykh, A. P., editors, *Proceedings of the Fourth International Workshop on Verification and Program Transformation, VPT@ETAPS 2016, Eindhoven, The Netherlands, 2nd April 2016.*, volume 216 of *EPTCS*, pages 104–117, 2016.

[98] Kafle, B. and Gallagher, J. P. Constraint specialisation in Horn clause verification. *Sci. Comput. Prog.*, 2016. (In press).

[99] Kafle, B., Gallagher, J. P., and Ganty, P. Decomposition by tree dimension in Horn clause verification. In Lisitsa, A., Nemytykh, A. P., and Pettorossi, A., editors, *VPT*, volume 199 of *EPTCS*, pages 1–14, 2015.

[100] Kafle, B., Gallagher, J. P., and Ganty, P. Solving non-linear horn clauses using a linear horn clause solver. In Gallagher, J. P. and Rümmer, P., editors, *Proceedings 3rd Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2016, Eindhoven, The Netherlands, 3rd April 2016.*, volume 219 of *EPTCS*, pages 33–48, 2016.

[101] Kafle, B., Gallagher, J. P., and Ganty, P. Solving non-linear Horn clauses using a linear Horn clause solver. *EPTCS*, 2016. (In press).

[102] Kafle, B., Gallagher, J. P., and Morales, J. F. Rahft: A tool for verifying horn clauses using abstract interpretation and finite tree automata. In Chaudhuri, S. and Farzan, A., editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part I*, volume 9779 of *Lecture Notes in Computer Science*, pages 261–268. Springer, 2016.

[103] Kafle, B., Gallagher, J. P., and Morales, J. F. RAHFT: A tool for verifying Horn clauses using abstract interpretation and finite tree automata. *Computer Aided Verification (CAV)*, 2016. (In press).

[104] Klimov, A. V. Solving coverability problem for monotonic counter systems by supercompilation. In Clarke, E. M., Virbitskaite, I., and Voronkov, A., editors, *Perspectives of Systems Informatics - 8th International Andrei Ershov Memorial Conference, PSI 2011, Novosibirsk, Russia, June 27-July 1, 2011, Revised Selected Papers*, volume 7162 of *LNCS*, pages 193–209. Springer, 2011.

[105] Klyuchnikov, I. G. and Romanenko, S. A. Proving the equivalence of higher-order terms by means of supercompilation. In Pnueli, A., Virbitskaite, I., and Voronkov, A., editors, *Perspectives of Systems Informatics, 7th International Andrei Ershov Memorial Conference, PSI 2009, Novosibirsk, Russia, June 15-19, 2009. Revised Papers*, volume 5947 of *LNCS*, pages 193–205. Springer, 2009.

[106] Komorowski, H. J. An introduction to partial deduction. In Pettorossi, A., editor, *META*, volume 649 of *LNCS*, pages 49–69. Springer, 1992.

[107] Lafave, L. and Gallagher, J. P. Partial evaluation of functional logic programs in rewriting-based languages. In Fuchs, N., editor, *Logic Program Synthesis and Transformation (LOPSTR'97)*, Springer-Verlag LNCS, 1998.

[108] Lakhdar-Chaouch, L., Jeannet, B., and Girault, A. Widening with thresholds for programs with complex control graphs. In Bultan, T. and Hsiung, P.-A., editors, *ATVA 2011*, volume 6996 of *LNCS*, pages 492–502. Springer, 2011.

[109] Launchbury, J. and Mitchell, J. C., editors. *Conference Record of POPL 2002: The 29th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, OR, USA, January 16-18, 2002*, 2002. ACM.

[110] Leuschel, M. Partial evaluation of the "real thing". In Fribourg, L. and Turini, F., editors, *Logic Programming Synthesis and Transformation - Meta-Programming in Logic., Proceedings*, volume 883 of *LNCS*, pages 122–137. Springer, 1994.

[111] Leuschel, M. Advanced logic program specialisation. In Hatcliff, J., Mogensen, T. Æ., and Thiemann, P., editors, *Partial Evaluation - Practice and Theory*, volume 1706 of *LNCS*, pages 271–292. Springer, 1999.

[112] Leuschel, M. A framework for the integration of partial evaluation and abstract interpretation of logic programs. *ACM Trans. Program. Lang. Syst.*, 26(3):413–463, 2004.

[113] Leuschel, M. and Massart, T. Infinite state model checking by abstract interpretation and program specialisation. In Bossi, A., editor, *LOPSTR'99*, volume 1817 of *LNCS*, pages 62–81. Springer, 1999.

[114] Leuschel, M. and Sørensen, M. H. Redundant argument filtering of logic programs. In Gallagher, J. P., editor, *Logic Programming Synthesis and Transformation, 6th International Workshop, LOPSTR'96, Stockholm, Sweden, August 28-30, 1996, Proceedings*, pages 83–103, 1996.

[115] Leuschel, M. and Vidal, G. Fast offline partial evaluation of logic programs. *Inf. Comput.*, 235:70–97, 2014.

[116] Leuschel, M., Elphick, D., Varea, M., Craig, S.-J., and Fontaine, M. The Ecce and Logen partial evaluators and their web interfaces. In Hatcliff, J. and Tip, F., editors, *PEPM 2006*, pages 88–94. ACM, 2006.

[117] Levi, G. Abstract interpretation based verification of logic programs. *Electr. Notes Theor. Comput. Sci.*, 40:243, 2000.

[118] Lisitsa, A. and Nemytykh, A. P. Reachability analysis in verification via supercompilation. *Int. J. Found. Comput. Sci.*, 19(4):953–969, 2008.

[119] Lloyd, J. W. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[120] Lloyd, J. *Foundations of Logic Programming: 2nd Edition*. Springer-Verlag, 1987.

[121] Luttenberger, M. and Schlund, M. Convergence of Newton's method over commutative semirings. *Inf. Comput.*, 246:43–61, 2016.

[122] Manna, Z. and Pnueli, A. Formalization of properties of functional programs. *J. ACM*, 17(3):555–569, 1970.

[123] Marriott, K., Naish, L., and Lassez, J.-L. Most specific logic programs. In *Proc. Fifth International Conference on Logic programming, Seattle, WA*. MIT Press, 1988.

[124] Marriott, K. and Stuckey, P. J. The 3 r's of optimizing constraint logic programs: Refinement, removal and reordering. In Deusen, M. S. V. and Lang, B., editors, *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Charleston, South Carolina, USA, January 1993*, pages 334–344. ACM Press, 1993.

[125] McMillan, K. L. Interpolants from Z3 proofs. In Bjesse, P. and Slobodová, A., editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 19–27. FMCAD Inc., 2011.

[126] McMillan, K. L. and Rybalchenko, A. Solving constrained Horn clauses using interpolation. Technical report, Microsoft Research, 2013.

[127] Monniaux, D. and Gonnord, L. Using bounded model checking to focus fixpoint iterations. In Yahav, E., editor, *Static Analysis - 18th International Symposium, SAS 2011, Venice, Italy, September 14-16, 2011. Proceedings*, volume 6887 of *LNCS*, pages 369–385. Springer, 2011.

[128] Nilsson, U. Abstract interpretation: A kind of magic. *Theor. Comput. Sci.*, 142(1):125–139, 1995.

[129] Peralta, J., Gallagher, J. P., and Sağlam, H. Analysis of imperative programs through analysis of constraint logic programs. In Levi, G., editor, *Static Analysis. 5th International Symposium, SAS'98, Pisa*, volume 1503 of *Springer-Verlag LNCS*, pages 246–261, 1998.

[130] Peralta, J. C. and Gallagher, J. P. Convex hull abstractions in specialization of CLP programs. In Leuschel, M., editor, *LOPSTR*, volume 2664 of *LNCS*, pages 90–108. Springer, 2002.

[131] Pettorossi, A. and Proietti, M. Synthesis and transformation of logic programs using unfold/fold proofs. *J. Log. Program.*, 41(2-3):197–230, 1999.

[132] Pettorossi, A. and Proietti, M. Perfect model checking via unfold/fold transformations. In Lloyd, J. W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Palamidessi, C., Pereira, L. M., Sagiv, Y., and Stuckey, P. J., editors, *Computational Logic*, volume 1861 of *LNCS*, pages 613–628. Springer, 2000.

[133] Piterman, N. and Smolka, S. A., editors. *TACAS. Proceedings*, volume 7795 of *LNCS*, 2013. Springer.

[134] Podelski, A. and Rybalchenko, A. ARMC: the logical choice for software model checking with abstraction refinement. In Hanus, M., editor, *PADL 2007*, volume 4354 of *LNCS*, pages 245–259, 2007.

[135] Ramakrishnan, R. Magic templates: A spellbinding approach to logic programs. In Kowalski, R. A. and Bowen, K. A., editors, *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August 15-19, 1988 (2 Volumes)*, pages 140–159. MIT Press, 1988.

[136] Rümmer, P., Hojjat, H., and Kuncak, V. Disjunctive interpolants for Horn-clause verification. In Sharygina and Veith [140], pages 347–363.

[137] Rümmer, P., Hojjat, H., and Kuncak, V. Disjunctive interpolants for Horn-clause verification. In Sharygina, N. and Veith, H., editors, *CAV 2013, Proceedings*, volume 8044 of *LNCS*, pages 347–363. Springer, 2013.

[138] Rybalchenko, A. and Sofronie-Stokkermans, V. Constraint solving for interpolation. *J. Symb. Comput.*, 45(11):1212–1233, 2010.

[139] Serebrenik, A. and De Schreye, D. Inference of termination conditions for numerical loops in Prolog. In Nieuwenhuis, R. and Voronkov, A., editors, *LPAR 2001*, volume 2250 of *LNCS*, pages 654–668. Springer, 2001.

[140] Sharygina, N. and Veith, H., editors. *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *LNCS*, 2013. Springer.

[141] Srivastava, D. and Ramakrishnan, R. Pushing constraint selections. *J. Log. Program.*, 16 (3):361–414, 1993.

[142] Stärk, R. F. A direct proof for the completeness of SLD-resolution. In Börger, E., Büning, H. K., and Richter, M. M., editors, *CSL '89, 3rd Workshop on Computer Science Logic, Kaiserslautern, Germany, October 2-6, 1989, Proceedings*, volume 440 of *LNCS*, pages 382–383. Springer, 1989.

[143] Tarjan, R. E. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2):146–160, 1972.

[144] Turchin, V. F. The use of metasystem transition in theorem proving and program optimization. In de Bakker, J. W. and van Leeuwen, J., editors, *Automata, Languages and Programming, 7th Colloquium, Noordweijkerhout, The Netherland, July 14-18, 1980, Proceedings*, volume 85 of *LNCS*, pages 645–657. Springer, 1980.

[145] Ullman, J. *Principles of Knowledge and Database Systems; Volume 1*. Computer Science Press, 1988.

[146] Vieille, L. Recursive axioms in deductive databases: The query/subquery approach. In *Expert Database Conf.*, pages 253–267, 1986.

[147] Wang, W. and Jiao, L. Trace abstraction refinement for solving Horn clauses. Technical Report ISCAS-SKLCS-15-19, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Dec. 2015. Available on: http://lcs.ios.ac.cn/~wangwf/TechReportISCAS-SKLCS-15-19.pdf.

[148] Winsborough, W. H. Multiple specialization using minimal-function graph semantics. *J. Log. Program.*, 13(2&3):259–290, 1992.

DECLARATION

I hereby declare that this PhD thesis entitled "Components for automatic Horn clause verification" was carried out by me for the PhD degree in Computer science under the guidance and supervision of Prof. John Gallagher, Institute of People and Technology, Roskilde University, Denmark.

I certify that this work contains no material which has been accepted for the award of any other degree or diploma in my name, in any university or other tertiary institution and, to the best of my knowledge and belief, contains no material previously published or written by another person, except where due reference has been made in the text. In addition, I certify that no part of this work will, in the future, be used in a submission in my name, for any other degree or diploma in any university or other tertiary institution without the prior approval of the Roskilde University and where applicable, any partner institution responsible for the joint-award of this degree.

Roskilde, October 31, 2016

Bishoksan Kafle, October 31, 2016